

ADA 216 407

Theoretical and Experimental Analyses  
of  
Parallel Combinatorial Algorithms

by

Cynthia A. Phillips

S.M. EECS, Massachusetts Institute of Technology (1985)  
B.A. Applied Mathematics, Harvard University (1983)

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 1989

© Massachusetts Institute of Technology 1989  
All rights reserved

Signature of Author Cynthia A. Phillips  
Department of Electrical Engineering and Computer Science  
October 1989

Certified by Charles E. Leiserson  
Charles E. Leiserson  
Associate Professor of Computer Science and Engineering  
Thesis Supervisor

Accepted by \_\_\_\_\_  
Arthur C. Smith  
Chairman, Departmental Committee on Graduate Students

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY JAN 3 1990			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-462		
5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-87-K-0825 N00014-86-K0593			6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science		
6b. OFFICE SYMBOL (If applicable)			7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy		
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139			7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD		8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.		PROJECT NO.	
		TASK NO.		WORK UNIT ACCESSION NO.	
11. TITLE (Include Security Classification) Theoretical and Experimental Analyses of Parallel Combinatorial Algorithms					
12. PERSONAL AUTHOR(S)					
13a. TYPE OF REPORT Technical		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Year, Month, Day) October 1989	
15. PAGE COUNT 153					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	parallel algorithms, combinatorial algorithms, graphs, graph contraction, connected components, linear algebra, matrices, hypercube, Connection Machine, load balancing,		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>This thesis investigates parallel algorithms for a small, but representative, subclass of graph and matrix problems. In some cases, we develop new algorithms which we analyze for theoretical efficiency. In other cases, we modify and implement existing algorithms which we analyze for practical efficiency.</p> <p>We show how <math>n</math>-node, <math>e</math>-edge graphs can be contracted in a manner similar to the parallel tree contraction algorithm due to Miller and Reif. We give an <math>O((n+e)/\lg n)</math>-processor deterministic algorithm that contracts a graph in <math>O(\lg^2 n)</math> time in the EREW PRAM model. We also give an <math>O(n/\lg n)</math>-processor randomized algorithm that with high probability can contract a bounded-degree graph in <math>O(\lg n + \lg^2 \gamma)</math> time, where <math>\gamma</math> is the maximum genus of any connected component of the graph. (The algorithm can be made to run in deterministic <math>O(\lg n \lg^2 n + \lg^2 \gamma)</math> time using known techniques.) This algorithm does not require a priori knowledge of the genus</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little			22b. TELEPHONE (Include Area Code) (617) 253-5894		22c. OFFICE SYMBOL

18. cont.

combinatorial optimization, assignment, Traveling Salesman Problem, permutation, networks, Benes network, VLSI.

19. cont.

of the graph to be contracted. The contraction algorithm for bounded-degree graphs can be used directly to solve the problem of region labeling in vision systems, i.e., determining the connected components of bounded-degree planar graphs in  $O(\lg n)$  time, thus improving the best previous bound of  $O(\lg^2 n)$ .

We also describe four APL-like primitives for manipulating dense matrices and vectors and describe their implementation on the Connection Machine<sup>1</sup> hypercube multiprocessor. These primitives provide a natural way of specifying parallel matrix algorithms independently of machine size or architecture and can actually enhance efficiency by facilitating automatic load balancing. The implementations are efficient in the frequently occurring case where there are fewer processors than matrix elements. In particular, if there are  $m > plgp$  matrix elements, where  $p$  is the number of processors, then the implementations of some of the primitives are asymptotically optimal for a weak hypercube in that the processor-time product is no more

---

<sup>1</sup>Connection Machine is a registered trademark of Thinking Machines Corporation

than a constant factor higher than the running time of the best serial algorithm. Furthermore, the parallel time required is optimal to within a constant factor. Our implementation of the primitives on the Connection Machine 2 system improved the performance of a simplex program for linear programming by almost an order of magnitude over a naive implementation, from 55 Mflops to 525 Mflops.

We investigate *dimension-exchange load balancing* which is a generalization of one of the techniques used in the hypercube implementation of the vector-matrix primitives. We show that when tasks are considered indivisible, after one pass of dimension-exchange load balancing, in the worst case, some processor will have  $\Theta(\lg n)$  tasks over the average. We also show that there is an initial distribution of tasks for which this load-balancing strategy requires an average of  $\Theta(\lg n)$  messages for each unit reduction in the global maximum number of tasks.

We report on preliminary experimental investigations which indicate that massively parallel computers like the Connection Machine (CM) appear to be well suited for both sparse and dense implementations of dual relaxation algorithms for network optimization. Implementations of a dense version of a known algorithm for the assignment problem and parallel versions of known heuristics for the traveling salesman problem suffered from a "sequential tail" phenomenon. Tail-cutting heuristics with appropriate (case-sensitive) parameters improved performance markedly.

We detail the design of a VLSI chip which pseudorandomly permutes bit-serial messages by sending them through a Benes network whose switches have been pseudorandomly set. Providing a pseudorandom permuter in a simple, high-throughput chip could improve the performance of routing algorithms for multiprocessors.

Theoretical and Experimental Analyses  
of  
Parallel Combinatorial Algorithms  
by  
Cynthia A. Phillips

Submitted to the Department of Electrical Engineering and Computer Science  
in October 1989, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

**Keywords:** parallel algorithms, combinatorial algorithms, graphs, graph contraction, connected components, linear algebra, matrices, hypercube, Connection Machine, load balancing, combinatorial optimization, assignment, Traveling Salesman Problem, permutation networks, Benes network, VLSI.

### Abstract

This thesis investigates parallel algorithms for a small, but representative, subclass of graph and matrix problems. In some cases, we develop new algorithms which we analyze for theoretical efficiency. In other cases, we modify and implement existing algorithms which we analyze for practical efficiency.

We show how  $n$ -node,  $e$ -edge graphs can be *contracted* in a manner similar to the parallel tree contraction algorithm due to Miller and Reif. We give an  $O((n+e)/\lg n)$ -processor deterministic algorithm that contracts a graph in  $O(\lg^2 n)$  time in the EREW PRAM model. We also give an  $O(n/\lg n)$ -processor randomized algorithm that with high probability can contract a bounded-degree graph in  $O(\lg n + \lg^2 \gamma)$  time, where  $\gamma$  is the maximum genus of any connected component of the graph. (The algorithm can be made to run in deterministic  $O(\lg n \lg^* n + \lg^2 \gamma)$  time using known techniques.) This algorithm does not require *a priori* knowledge of the genus of the graph to be contracted. The contraction algorithm for bounded-degree graphs can be used directly to solve the problem of region labeling in vision systems, *i.e.*, determining the connected components of bounded-degree planar graphs in  $O(\lg n)$  time, thus improving the best previous bound of  $O(\lg^2 n)$ .

We also describe four APL-like primitives for manipulating dense matrices and vectors and describe their implementation on the Connection Machine<sup>1</sup> hypercube multiprocessor. These primitives provide a natural way of specifying parallel matrix algorithms independently of machine size or architecture and can actually enhance efficiency by facilitating automatic load balancing. The implementations are efficient in the frequently occurring case where there are fewer processors than matrix elements. In particular, if there are  $m > p \lg p$  matrix elements, where  $p$  is the number of processors, then the implementations of some of the primitives are asymptotically optimal for a weak hypercube in that the processor-time product is no more

---

<sup>1</sup>Connection Machine is a registered trademark of Thinking Machines Corporation

than a constant factor higher than the running time of the best serial algorithm. Furthermore, the parallel time required is optimal to within a constant factor. Our implementation of the primitives on the Connection Machine 2 system improved the performance of a simplex program for linear programming by almost an order of magnitude over a naive implementation, from 55 Mflops to 525 Mflops.

We investigate *dimension-exchange load balancing* which is a generalization of one of the techniques used in the hypercube implementation of the vector-matrix primitives. We show that when tasks are considered indivisible, after one pass of dimension-exchange load balancing, in the worst case, some processor will have  $\Theta(\lg n)$  tasks over the average. We also show that there is an initial distribution of tasks for which this load-balancing strategy requires an average of  $\Theta(\lg n)$  messages for each unit reduction in the global maximum number of tasks.

We report on preliminary experimental investigations which indicate that massively parallel computers like the Connection Machine (CM) appear to be well suited for both sparse and dense implementations of dual relaxation algorithms for network optimization. Implementations of a dense version of a known algorithm for the assignment problem and parallel versions of known heuristics for the traveling salesman problem suffered from a "sequential tail" phenomenon. Tail-cutting heuristics with appropriate (case-sensitive) parameters improved performance markedly.

We detail the design of a VLSI chip which pseudorandomly permutes bit-serial messages by sending them through a Benes network whose switches have been pseudorandomly set. Providing a pseudorandom permuter in a simple, high-throughput chip could improve the performance of routing algorithms for multiprocessors.

Thesis Supervisor: Charles E. Leiserson

Title: Associate Professor of Computer Science and Engineering



<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
<b>Availability Codes</b>	
Dist	Avail and/or Special
A-1	

# Acknowledgements

I was warned before I came to MIT that it could be a cold and lonely place. I am happy to say that in the theory of computing (TOC) group, nothing could be further from the truth. As the end of the road is finally in sight, I can now begin one of the most enjoyable tasks: thanking everyone who made this thesis possible and made my graduate experience more pleasant.

First and foremost, I want to thank my advisor Charles Leiserson for all his help. There is absolutely no doubt in my mind that I never would have finished without his support and advice every inch of the way. I'm grateful for his technical and literary guidance, our many collaborations and for his opening the doors to Thinking Machines Corporation. I'm grateful for the financial support for travel, RA's, and the loan of the Macintosh that this thesis was finally cranked out on. Perhaps most importantly I am grateful for his confidence in me when I had none in myself, for his understanding and support when I found myself in the dual role of parent and graduate student, and for the uncountably many little kindnesses.

I'd like to thank all my officemates through the years for their companionship, advice, and secretarial services: Torben Hagerup, Benny Chor, Andrew Goldberg, Jon Riecke, Eric Schwabe, and Alex Ishii.

I'd like to thank everyone who helped in the development and presentation of the graph contraction results of chapter 2. The development of the contraction algorithm for bounded degree graphs was joint work with Charles Leiserson. Thanks also to Gary Miller of USC and Miller Maley of Princeton for general assistance on topology issues, to Washington Taylor of Thinking Machines Corporation who provided the key topological ideas in the proof of lemma 6, and to James Park, Tom Cormen, and Bruce Maggs of MIT for their help with the figures and text formatting.

The vector-matrix primitives presented in chapter 3 were developed and implemented as joint work with Ajit Agrawal (now at Brown), Guy Blelloch (now at CMU), and Robert Krawitz while we were all at Thinking Machines Corporation. Lennart Johnsson of Thinking Machines Corporation and Yale gave us commentary on early drafts of our writeup and provided us with a list of relevant references.

The network optimization implementations of chapter 5 are joint work with Stavros Zenios of the Wharton School at U. Penn. The traveling salesman implementation described in chapter 6 is joint work with Ron Greenberg and Joel Wein of MIT. The chip design described in chapter 7 was inspired by Charles Leiserson. I also relied on conversations with Ron Rivest, Alan Sherman, Miller Maley, and Ray Hirschfeld in making design decisions for the pseudorandom number generator.

I'd like to thank everyone I worked with at Thinking Machines Corporation from the summer of 1987 to now. I felt like I was on the cutting edge of technology (which I was), surrounded by

energetic, knowledgeable, talented people. I was impressed that nobody was ever too busy to answer a question without making me feel like a fool or an imposition, and everyone knew how to have a good time occasionally. Thanks to Lennart Johnsson, Xiru Zhang, Joel Wein, Bruce Boghosian, Robert Krawitz, and all the other members of MCSG, and very special thanks to Jill Mesirov, Carmen Crocker, and Luis Ortiz for help above and beyond the call of duty.

I'd like to thank everyone in the theory of computing group for all the help and fun through the last six years, especially everyone who watched Tommy for me during the last rushed year (Dina, Be, Aditi, Sally G, Joe) and all the members of the ice hockey and softball teams for all the great games. Thanks to Sally Bemus and Ray Hirschfeld for all the systems help. Thanks to Albert Meyer for letting me know about the IBM fellowship and helping me get it. Thanks to Shlomo Kipnis for arranging for a flight back from Albuquerque (even though I didn't make it back). A Huge thanks to Joel Wein for going through my unintelligible files, under very inconvenient circumstances, to ship out data I needed in the last week of writing. Thanks to James Park, Tom Cormen, Bruce Maggs, Eric Schwabe, Alex Ishii, and Joe Kilian for making me laugh during even the worst of times. Thanks to Jennifer Welsh and Sally Goldman for the commiseration lunches and the long conversations when I needed a sympathetic ear or parenting advice. Thanks to Be Hubbard (aka. Mom) for things too numerous to mention: phone calls and forms and errands and conversation and support and everything a mom would do for one of her kids.

Thanks to Marilyn Pierce for guiding me through the departmental red tape and for keeping a concerned eye on me (for example, making sure that I was registered for the last semester).

Thanks to my committee: Charles Leiserson, Tom Knight, and David Shmoys. It is never easy to serve on a thesis committee and it is especially difficult when things must be finished at long distance. Thanks especially to David Shmoys for agreeing to come back from Cornell for the defense. Thanks also to my graduate counselor Sylvio Micali who inherited me when I was well along in the process and being hounded by the grad office for delinquency.

I was supported by the Defense Advanced Research Projects Agency under Contract N00014-87-K-0825 and by the Office of Naval Research under Contract N00014-86-K-0593. I also am grateful to IBM for their support as a graduate fellow from 1987 to 1989.

Finally, I'd like to thank my family. I thank my parents for their financial support and encouragement. Mostly I thank Tom for being there every day, putting up with all my moods and frustrations, for cooking and watching Tommy in the evenings to allow me to finish up, for his UNM ID which got me into the laser printer facilities, and for his love and confidence.

# Contents

<b>1 Preliminaries</b>	<b>12</b>
1.1 Parallel Models of Computation . . . . .	12
1.1.1 The Distributed Model . . . . .	12
1.1.2 The PRAM Family of Models . . . . .	13
1.1.3 The Data Parallel Model . . . . .	15
1.2 Interconnection Networks . . . . .	16
1.3 The Connection Machine System . . . . .	17
<b>2 Parallel Graph Contraction</b>	<b>21</b>
2.1 Introduction . . . . .	21
2.2 The Graph Data Structure . . . . .	22
2.3 General Algorithm . . . . .	24
2.4 Bounded-Degree Algorithm . . . . .	26
2.5 Analysis of the Contraction Algorithm . . . . .	28
2.6 Missing-Edge Lemma . . . . .	33
2.7 Applications . . . . .	39
2.8 Conclusion . . . . .	40
<b>3 Vector-Matrix Primitives</b>	<b>42</b>
3.1 Introduction . . . . .	42
3.2 Example Applications . . . . .	48
3.2.1 Matrix-Vector Multiply . . . . .	48
3.2.2 Linear System Solution by LU Decomposition . . . . .	49
3.2.3 Simplex method for linear programming . . . . .	51
3.3 Implementation of Primitives . . . . .	53
3.3.1 Extract . . . . .	54
3.3.2 Reduce . . . . .	56
3.3.3 Spreads . . . . .	60
3.3.4 Analysis . . . . .	62
3.3.5 Computing on a Single Row or Column . . . . .	65
3.3.6 Extensions . . . . .	66
3.4 Timings . . . . .	67
3.5 Conclusions . . . . .	68

<b>4</b>	<b>Dimension-Exchange Load Balancing</b>	<b>70</b>
4.0.1	Upper Bounds	72
4.0.2	Lower Bounds	75
4.0.3	Message Complexity	83
4.1	Conclusions	85
<b>5</b>	<b>The Assignment Problem</b>	<b>86</b>
5.1	Introduction	86
5.2	Representing Networks on the Connection Machine	89
5.2.1	Representation of Dense Assignment Problems	89
5.2.2	Representation of Sparse Network Problems	89
5.3	Bertsekas' Algorithm on the Connection Machine	91
5.4	Tails and Tail Cutting	92
5.5	Experimental Results	94
5.6	Summary and conclusions	101
<b>6</b>	<b>Parallel Traveling Salesman Heuristics</b>	<b>105</b>
6.1	Introduction	105
6.2	Building the Initial Tour	107
6.2.1	Greedy Heuristics	107
6.2.2	Partitioning Heuristics	109
6.3	Tweaking	110
6.4	Results and Conclusions	113
<b>7</b>	<b>Pseudorandom Permuter Chip</b>	<b>120</b>
7.1	Definition and Purpose	121
7.2	The Permutation Network	122
7.3	Pseudorandomly Setting Bits	123
7.4	Architecture	126
7.5	Operations	130
7.5.1	External Control	130
7.5.2	Load Poly Register (op code 00)	132
7.5.3	Load a Switch Row (op code 01)	133
7.5.4	New Permutation (op code 10)	133
7.5.5	Send Messages (op code 11)	133
7.5.6	Switching Out of Message Sending Mode	134
7.6	Logic Design and Layout	134
7.6.1	Overall concerns	134
7.6.2	Implementation of Individual Units	137
7.6.3	Problems with the Layout	143

# Introduction

Those faced with a huge computational task that must be accomplished in real time, or an astronomical task that must be performed in moderate time, are turning increasingly to massively parallel computers. Although the technology is promising, it is still too early to tell to what extent the attention and enthusiasm directed at parallel computing is justified. We still do not know the best way to build parallel machines. Although there is a growing body of theoretical research on parallel algorithms, there is a relative dearth of parallel systems tools and practical programming experience. The systems and architectural experience from VLSI design — massively parallel processing on the microscopic level — and algorithmic and linguistic techniques from sequential computing cannot be trivially generalized.

When faced with a largely unexplored domain with so much potential for complication, we are motivated by the following principle: *whenever possible, keep things simple*. Thus we aim for algorithmic, linguistic, and high- and low-level architectural simplicity without losing sight of our ultimate goal of performance. We measure performance in two ways: theoretical asymptotic performance relative to a realistic model of parallel computation, and actual performance as observed from implementations on actual parallel machines. The two goals are not incompatible. In some cases we can achieve performance because of simplicity. For example, simple algorithms and high-level languages lead to high programmer productivity and simple hardware components can lead to fast clocking and high throughput. In the best cases we can also achieve theoretically good performance in spite of simplicity. Frequently, algorithms that are theoretically good are initially complicated and simplicity comes later when we have acquired a greater understanding of the problem.

In the thesis, we concentrate on a set of *combinatorial* problems most of which can be formalized using graphs and/or matrices. This class of problems is particularly suited for parallel study from both a theoretical and practical standpoint. Because they offer a flexible representation of relationships among finite sets of objects, graphs and matrices are good models of real phenomena. There are many real instances of these problems in a wide range of application areas including defense planning, vision systems, operations research, transportation, engineering design, financial planning, and artificial intelligence. The problems that people actually want to solve are so large that they are infeasible on all but the largest supercomputers, and as computing resources increase, the problem sizes will grow too. For example, people will perform more fine-grained simulations or model larger systems. Because graphs and matrices are clean, mathematical objects, they are well suited for theoretical study. Because of the large problem size in practice, these asymptotic analyses are likely to be good indicators of actual performance, provided the models used for the analyses are sufficiently realistic.

We address a large number of concerns for a large class of problems, and therefore the

results of this thesis are not all directly related. Each chapter is self-contained and can be read independently. Virtually all achieve our goals to some extent by presenting some combination of simple algorithmic, linguistic, and architectural techniques which offer theoretical and/or heuristic parallel speedup.

The remainder of the thesis is organized as follows. Chapter 1 gives background on the models used in the theoretical analyses of this thesis, defines the relevant interconnection networks, and describes the Connection Machine hypercube multiprocessor which is the parallel system used for all experimental studies in this thesis.

Chapter 2 presents simple algorithms which *contract* graphs in a manner similar to the parallel tree contraction algorithm due to Miller and Reif. Suppose, for example, that each processor of a parallel machine contains one pixel of an image downloaded from a digital camera. Each processor knows only a set of neighboring processors that have the same color. Suppose we want each object in the picture to be displayed in a unique color. Then each processor must know to which object it belongs. If we represent the pixels as nodes of a graph and connect two processors by an edge if and only if they represent neighboring pixels of the same color, then the problem reduces to the *connected components* problem.

The technique of graph contraction is illustrated in figure 0-1. We start with an arbitrary graph and group sets of nodes that are known to be connected into *super nodes*. One can think of one node taking the edge connecting it to a neighbor and "pulling" the neighbor into itself until the edge is of length 0 and the two nodes are one. In the figure, we contract only single edges. An external observer looking at the contracted graph sees only the section shown in heavy lines on each step. The supernodes, however, maintain the illustrated inner structure so that the process can be reversed, thus restoring the old graph with knowledge of the globally connected components. The contraction process ends when all nodes from the same component are inside a single supernode.

We present a contraction algorithm for bounded-degree graphs (or general graphs given an embedding) which is asymptotically superior to any previous connected components algorithm for the EREW PRAM model, the most easily implemented of the standard PRAM models. More precisely, we give an  $O((n + e)/\lg n)$ -processor deterministic algorithm that contracts a graph in  $O(\lg^2 n)$  time in this model. We also give an  $O(n/\lg n)$ -processor randomized algorithm that with high probability can contract a bounded-degree graph in  $O(\lg n + \lg^2 \gamma)$  time, where  $\gamma$  is the maximum genus of any connected component of the graph. (The algorithm can be made to run in deterministic  $O(\lg n \lg^* n + \lg^2 \gamma)$  time using known techniques.) This algorithm does not require *a priori* knowledge of the genus of the graph to be contracted. These are the best times to date for bounded-degree graphs of moderate genus, namely  $\gamma = o(n^\epsilon)$  for constant  $\epsilon > 0$ . For the imaging problem we mentioned earlier, the graph arising from the pixels is planar and of bounded degree. Therefore, we can find the connected components in  $O(\lg n)$  time, thus improving the best previous bound of  $O(\lg^2 n)$ .

Chapter 3 describes four APL-like primitives for manipulating dense matrices and vectors and describes their implementation on the Connection Machine. These primitives provide a simple, natural way of specifying parallel matrix algorithms independently of machine size or architecture and can actually enhance efficiency by facilitating automatic load balancing. These simple language primitives have a simple implementation on a hypercube machine. Not only did the implementations give tremendous experimental speedup of applications running on the Connection Machine, but also they are theoretically efficient in the frequently occurring case

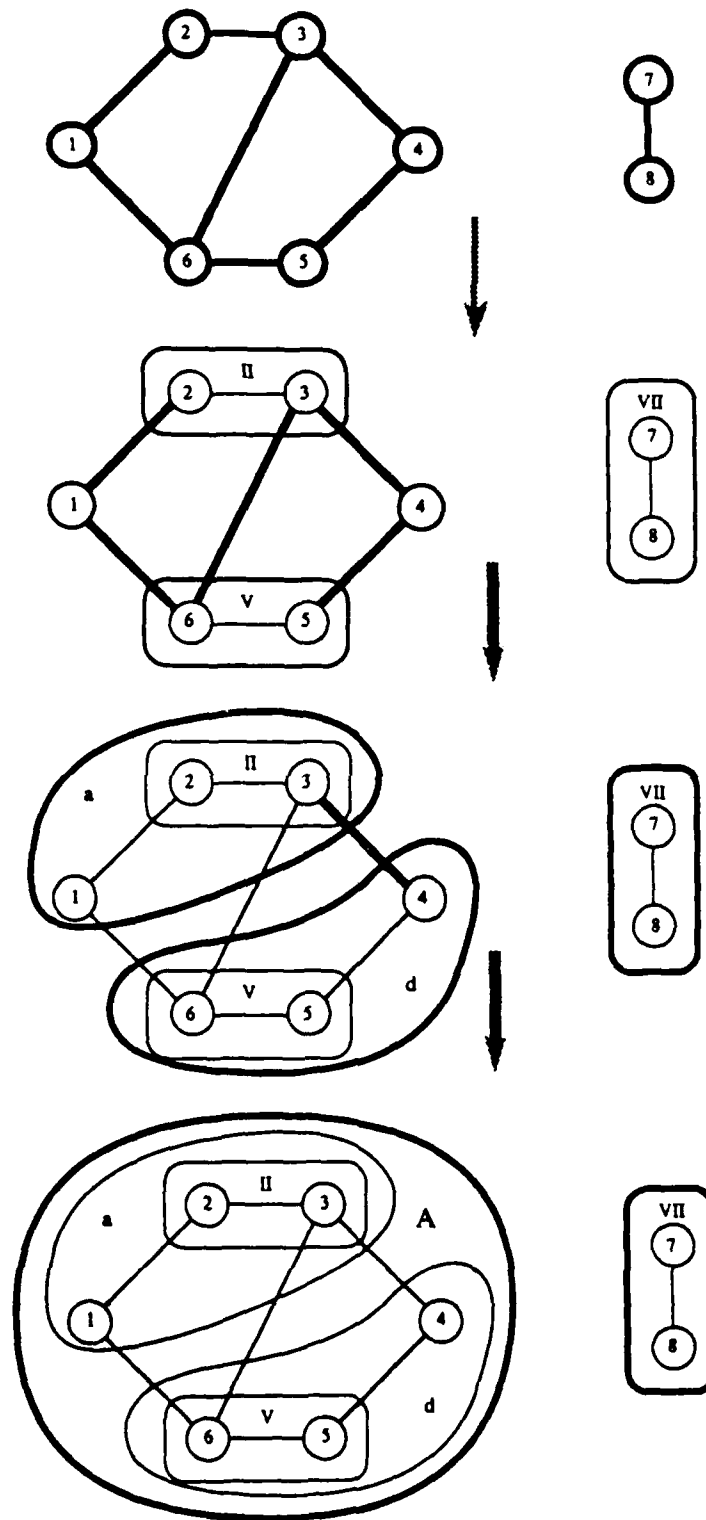


Figure 0-1: During graph contraction, neighboring nodes are grouped into supernodes, by conceptually "contracting" their connecting edges, until each component is an isolated supernode. The graph resulting from each successive contraction step is shown in heavy lines.

where there are fewer processors than matrix elements.

Chapter 4 considers a load-balancing technique that is a generalization of the hypercube implementation of the *extract* primitive presented in chapter 3. For any distribution of computations on a hypercube, we can balance the load by proceeding through each dimension and balancing the load between neighbors in that dimension. Cybenko [32] calls this technique the *dimension-exchange* method and he proves that it is superior to a *diffusion* method where each processor averages the work among all neighbors on each step. Cybenko's analysis allows tasks to be infinitely divisible. We consider the case where computations are indivisible, as they are in practice when the tasks are primitive operations. In particular we show that for a single pass through all dimensions, the processor with the most elements will have  $\Theta(\lg n)$  elements over the average where  $n$  is the number of processors in the hypercube. We can achieve the upper bound by simply dividing equally among neighbors with any extra tasks going to the processor whose identifier has even parity. We present a lower-bound proof which suffices for any oblivious dimension-exchange strategy which divides work evenly (within one). In this context, *oblivious* means that two processors use strictly local information when dividing work. Finally, we consider the benefits of reduced load vs. the added message complexity of the load-reduction process. In particular we show that there is a distribution of work such that the load balancing algorithm sends  $\Omega(w \lg n)$  messages to achieve a work reduction of  $O(w)$ .

Chapters 5 and 6 report preliminary results of implementations of several optimization algorithms. In these chapters we use simple heuristics to obtain experimental speedup. Chapter 5 describes a dense implementation of the assignment problem and chapter 6 describes a sparse parallel implementation of known heuristics for the NP-complete traveling salesman problem. Both of these implementations involve iteratively improving a feasible solution until some stopping criterion is met. For the case of the polynomial-time assignment algorithm, we are not supposed to stop until we reach optimality. The iterative improvement in both cases suffered from a "sequential tail" phenomenon, meaning that most of the parallelism occurred early, and the final fine-tuning became almost, if not entirely, sequential. The Connection Machine achieves its advantage by performing many parallel operations using simple, slow processors. This sequential tail could therefore be devastating for a Connection Machine implementation. Tail-cutting heuristics with appropriate (case-sensitive) parameters improved performance markedly without greatly hurting the quality of the output.

Chapter 7 differs somewhat from the preceding chapters because it describes a hardware solution to a more classical combinatorial problem. This chapter details the design of a VLSI chip, manufactured through MOSIS, which pseudorandomly permutes bit-serial messages by sending them through a Benes network whose switches have been pseudorandomly set. Providing a pseudorandom permuter in a simple, high-throughput chip could improve the performance of routing algorithms for multiprocessors. .

# Chapter 1

## Preliminaries

In this chapter we give background on theoretical models and actual machines used in this thesis. Section 1.1 describes some theoretical models that have been proposed for parallel computers, highlighting those models used in the analysis of the algorithms described in this thesis. In particular it defines the popular *parallel random access machine* (PRAM) and discusses the relative power of its subclasses. Section 1.2 lists some of the interconnection networks used to build actual parallel computers and defines the two of relevance to this thesis: the hypercube and the Benes network. Finally, section 1.3 describes the features of the Connection Machine, the parallel computer used for all implementations described in this thesis.

### 1.1 Parallel Models of Computation

In this section we describe briefly some of the models used in analyzing the performance of parallel algorithms. The major models of interest are the distributed model, the parallel random access machine (PRAM) family of models and the data-parallel model. We explain where the models differ and indicate which models are used in the various analyses of this thesis.

#### 1.1.1 The Distributed Model

In the distributed model of computation,  $n$  processors are interconnected by some two-point interconnection network. That is, the network forms a graph  $G = (V, E)$  where the vertices  $V$  are the processors and edge  $(u, v) \in E$  if and only if there is a communication channel connecting processors  $u$  and  $v$ . Usually, the network is restricted to bounded degree. In one step, each processor can send messages to one or more of its neighbors, but no communication can occur between non-neighbors.

For most algorithms, each processor requires some information from each other processor before it can make any final decisions. In this case the diameter of the network (i. e., the largest distance separating two processors) is a lower bound on the number of steps required to correctly execute the algorithm. For currently-realizable networks (for example, bounded-degree networks) the diameter is at least  $\Omega(\lg n)$  and it can be as large as  $n$ .

The running time in the distributed model is an appropriate complexity measure for machines such as simple mesh-connected processors where processors may have to communicate

over long paths. Our analyses of the load-balancing algorithms described in section 3 fit the distributed model in that we assume that communication occurs only along hypercube wires. Much of the research in the field of *distributed computing* is dedicated to issues related to networks of asynchronous processors, for example how to reach agreement, fault tolerance, clocking, etc. We consider all networks in this thesis, however, to be synchronous, and therefore our focus and analyses do not have the flavor of standard distributed arguments.

### 1.1.2 The PRAM Family of Models

In the parallel random access machine (PRAM) model,  $n$  processors are all connected to a common random access memory. Equivalently, each processor owns some local storage which all other processors can access. On a single step, each processor can read or write a location in the common memory and perform a simple arithmetic operation. This model was first formalized by Fortune and Wyllie [41]. Other early descriptions of PRAMs can be found in [51,59,100,108]. The three main types of PRAMs are distinguished by the way they handle read and write conflicts. In the exclusive-read exclusive-write (EREW) PRAM no simultaneous access of any kind is allowed. In the concurrent-read exclusive-write (CREW) PRAM, any number of processors can read a single location at the same time, but no two processors can write to the same location at the same time. Finally in the concurrent-read concurrent-write (CRCW) PRAM any number of processors can write to the same location at the same time. Many write conflict resolution policies appear in the literature including: the highest priority processor wins (*priority*), an arbitrary processor wins (*arbitrary*), all processor writing a location must write the same value (*common*), or detectable noise is written.

Other variations of the PRAM model have appeared in the literature. We mention just a few of note. In the family of *owner-write* models (EROW, CROW) introduced by Dymond and Ruzzo [36], each memory cell is owned by a processor and only the owner is allowed to write that cell. In the *scan* model introduced by Blelloch [14], parallel prefix computations can be performed in unit time. The *distributed random access machine* introduced by Leiserson and Maggs [78], as its name implies, captures some of the essential features of both the PRAM family of models and the distributed model. It is like the EREW PRAM model except that it assumes abstractly that there exists some underlying interconnection network. In this model there is a penalty for communications patterns that force a large number of messages across a small-bandwidth cut of the underlying intercommunication network. In practice, algorithms developed for the DRAM do not assume a specific interconnection network, but instead employ techniques that guarantee good communication behavior provided the initial problem is embedded well on the real machine.

The two main complexity measures in PRAM algorithms are the number of steps (time  $T$ ) and the number of processors  $P$ . A parallel algorithm has *optimal speedup* if the product  $PT$  equals the running time of the best known sequential algorithm. This definition stems in part from the inherent tradeoff between number of processors and running time. For example an algorithm that runs in time  $T$  using  $P$  processors can use half the number of processors if the time is doubled. Each processor simulates two. Thus in any of the parallel algorithms discussed, we can reduce the number of processors at the cost of a little more time.

From a complexity theoretic viewpoint, the three main types of PRAM form a strict hierarchy with the CRCW the most powerful, the CREW the next most powerful, and finally

the EREW the most restrictive. In terms of simulating each other, the gap is not large. A PRAM can trivially simulate any PRAM lower in the hierarchy. Thus any algorithm that runs in time  $T$  on an EREW will run on at most time  $T$  on a CREW or CRCW PRAM. Similarly, any algorithm that runs in time  $T$  on a CREW PRAM will require at most time  $T$  to run on a CRCW PRAM. Simulating a more powerful PRAM is also not too costly. Summarizing results of Eckstein [37] and Vishkin [123], Karp and Ramachandran show how any algorithm that runs in  $T$  steps on a CRCW PRAM with  $P$  processors can be simulated by a  $P$ -processor EREW PRAM using  $T \lg P$  steps [69].

Considerable investigation of the theoretical power of concurrency has led to the separation of the PRAM classes into the hierarchy described above. To demonstrate that concurrent write is more powerful than exclusive write, Cook, Dwork, and Reischuk [28] proved that the OR function on  $n$  bits requires  $\Omega(\lg n)$  time on a CREW PRAM even if that PRAM has unlimited processors, unlimited memory and memory wordsize, and unlimited computational power on each step. We can see, however, that a common CRCW PRAM can compute the OR in constant time with  $n$  processors each having constant number of bounded-sized memory cells. We simply initialize the first memory cell to 0 and have the  $i$ th processor write a 1 into the first memory cell if input bit  $i$  is 1.

A similar theoretical gap exists for concurrent reads. Snir [112] showed that range searching in a sorted list of size  $n$  requires  $\Theta(\sqrt{\lg n})$  time on any EREW PRAM, but only constant time on a CREW PRAM. Because the input list is assumed to be sorted, the input domain is not complete and the CREW PRAM can take advantage of this special knowledge. Gafni, Naor, and Ragde [44] found a function with complete domain that separates the EREW and CROW models. Recently, Fich and Widgerson [39] describe a boolean decision tree problem with a complete domain which separates the EREW PRAM from the CROW PRAM.

There does not appear to be a large theoretical difference between owner write and exclusive write. Nisan [92] proved that any boolean function with a complete domain has the same time complexity on CREW and CROW PRAMs to within a small constant factor. The Nisan result may seem a bit surprising in light of the "separation" shown by Cook, Dwork, and Reischuk [28]. They show that the OR of  $n$  variables containing at most one 1 requires  $\Omega(\lg n)$  time on a CROW PRAM, but only constant time on a CREW PRAM. For this problem, however, the domain is not complete and the CREW PRAM can use the special knowledge.

For more discussions of models, see [27,78,124]. Karp and Ramachandran [69] also survey results that yield a hierarchy among the CRCW PRAMs according to write-conflict resolution policies.

The PRAM model abstracts away the details of communication, allowing instead the comparison of inherent parallelism of algorithms. The model is indicative of the reasoning one uses in high-level programming of a machine, such as the Connection Machine, which has a general-purpose routing network. In practice, however, someone has to worry about how communication is actually performed. In particular, the processors must be connected by a physically realizable network such as a bounded-degree network or a bounded-size hypercube. Thus to understand the practicality of any algorithm designed for a PRAM, we must understand how that algorithm will be simulated on a real network and in particular, how packets will be routed to their destinations.

The most impressive recent packet routing result is Ranade's routing algorithm [102] which routes any set of  $n$  packets (including many-one) to destinations in time  $O(\lg n)$  on an  $n$ -node

butterfly (or hypercube) network with constant-size queues. The time bound is asymptotically optimal since any routing algorithm, even for permutations, requires at least the diameter of the network, and any practical network has diameter at least  $\Omega(\lg n)$ . Leighton, Maggs, and Rao [76] have recently achieved similar results for a more general class of networks including the shuffle-exchange graph and the fat-tree. Ranade builds upon earlier work by Aleliunas [2] and Upfal [119] who achieve similar results except that they require  $\Theta(\lg n)$ -size queues. Ranade's result seems to nullify the distinction between EREW and CRCW PRAMs in practice since either can be simulated in optimal  $O(\lg n)$  time with the same number of processors. Ranade's algorithm, however, is probabilistic and the need for combination of messages complicates the switches. The best deterministic simulations to date require time  $O(\lg^2 n / \lg \lg n)$  time, such as Hornick and Preparata's simulation on a two-dimensional mesh of trees [62] which achieves  $O(1)$  redundancy of variables at a cost of extra hardware. Specifically, for a PRAM with  $n$  processors, the mesh of trees used for the simulation has  $O(n^{2+\delta})$  processors for a constant  $\delta > 0$ . If the memory of the PRAM is sufficiently large, however, the VLSI area of their implementation is comparable with the area required by the PRAM. See [62] and [76] for surveys of previous packet routing results.

In this thesis, the algorithms for graph contraction are all designed for the EREW PRAM. We have seen that it is theoretically the most restrictive of the main PRAM models. In practice it is also the most easily implemented in hardware, although the practical effect of added combining hardware on the performance of routers is still not completely understood.

### 1.1.3 The Data Parallel Model

One of the most innovative architectures of parallel computer systems is the Connection Machine of Hillis [57] described in more detail in section 1.3. While the majority of parallel systems aim at improved computational performance with high-performance circuitry, coupled with vector features and a small (4) to medium (64) number of processors, the Connection Machine provides a design of up to 64K (=65,536) very simple processors coordinating on the solution of a single problem. The availability of this system in hardware (models CM-1 and CM-2) made necessary a new way of looking into numerical and other algorithms. Instead of examining the algorithm for possible parallel execution of control structures, CM programming introduces the notion of *data level parallelism* where a processor is assigned to each piece of data and similar operations are performed on all data in parallel.

In a *data parallel architecture*, each processor executes the same instruction broadcast from a *front end* computer. All processors access the same local memory address on each instruction. In such architectures, the effect of multiple instructions is achieved by allowing any processor to temporarily ignore the instruction stream based upon local variables. The set of processors which execute an instruction is called the *active set*. The data parallel architecture is sometimes called the *broadcast instruction* architecture or the *single instruction multiple data* (SIMD) model of parallel computation. The STARAN [6], MPP [99], and Connection Machine [57] all use this type of instruction stream. The data-parallel model was first introduced by Hillis and Steele [58].

The vector-matrix primitives and the network optimization implementations were programmed on the Connection Machine and therefore can be analyzed in the data parallel model. The graph contraction algorithm assigns a processor to each edge of a graph and therefore, also

fits well into the data parallel model. An algorithm need not be designed with the Connection Machine, or a similar architecture, in mind to be data parallel. In fact, many algorithms designed for the PRAM can be ported to the data parallel model with no asymptotic loss of speed, particularly if the data parallel model is expanded to allow indirect addressing of memory. This is because, although the PRAM allows unbounded control parallelism, most algorithms designed in practice have only a constant number of threads of control at any given time. Such algorithms can be run with only a constant slowdown on a data parallel machine by multiplexing over the control streams.

## 1.2 Interconnection Networks

A variety of interconnection networks (INs) have been proposed and/or built for parallel computers. There is generally a tradeoff between a network's wire area requirements and its ability to route arbitrary message patterns. The simplest INs are buses, linear arrays, and meshes. Buses can have congestion problems and meshes have a high diameter. Universal networks which are capable of simulating PRAMs with polylogarithmic slowdown include the hypercube, butterfly (also called the Omega or FFT), cube-connected-cycles, Benes, and shuffle-exchange. The fat-tree [77] and the mesh of trees [75] have been proven to be *area and volume universal* meaning that they can simulate any network of comparable area or volume with only polylogarithmic slowdown. See [118] for more details on universal routing networks. In this section we discuss the two networks of relevance to the thesis: the hypercube and the Benes network.

*Hypercube* multiprocessors have  $N = 2^n$  processors. Each processor has a unique address which is  $\lg N = n$  bits long and each processor is directly connected to all processors whose address differs from its own in exactly one bit position. Two connected processors whose address differs in bit  $i$  are called  *$i$ th-dimensional neighbors*. Processors in a *strong* hypercube multiprocessor can communicate with all  $\lg n$  neighbors simultaneously. Processors in a *weak* hypercube multiprocessor can communicate across only one dimension at a time, the same dimension in all processors. This thesis includes implementations of the vector-matrix primitives for a weak hypercube multiprocessor.

We now describe the *Benes* network used on the pseudorandom permuter chip. Figure 1-1 illustrates what I will call a *primitive switch* and later just a *switch* for short. It is a box with two inputs and two outputs. The inputs are routed to distinct outputs as directed by a single control bit. Informally, the inputs travel "straight" if the control bit is 0 and they "cross" if the control bit is 1.

The Benes network is composed entirely of primitive switches, can realize any permutation of its inputs without collisions, and the resulting permutation depends only upon the control bits of the basic switches. The network was constructed by Abraham Waksman [125]. It is best defined pictorially. Figure 1-2 is more or less a copy of the figure Waksman used to define the network in his original paper [125]. Each of the small boxes with an "S" in it is a primitive switch. The  $n$  inputs are paired and fed into the input of  $n/2$  primitive switches. The  $n$  outputs emerge in pairs from  $n/2$  primitive switches. The boxes labeled  $P_A$  and  $P_B$  represent permutation networks on  $n/2$  inputs formed recursively. The recursion will stop at the case  $n = 2$  with the primitive switch. This recursive definition restricts Benes networks to those cases where  $n$  is a power of 2, but it has the advantage of being regular and easy to understand. For those who are familiar with the butterfly network, one can think of the Benes network as

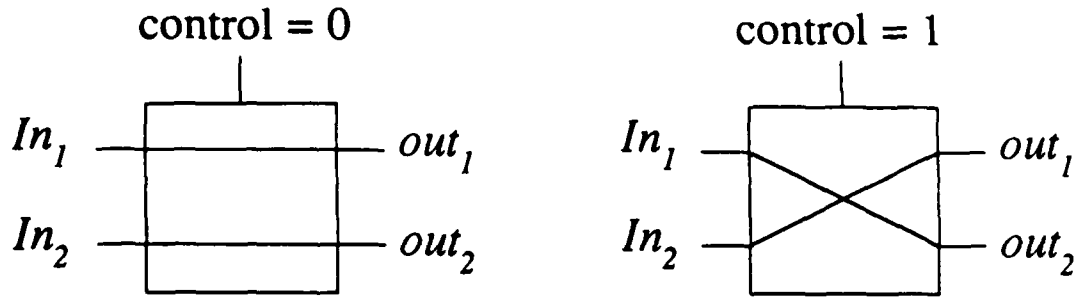


Figure 1-1: A primitive switch. The inputs are routed to outputs based upon the value of the control bit.

two butterfly networks back to back with the second one reflected.

When this definition is expanded to obtain the actual network, each column has exactly  $n/2$  primitive switches. The first and last columns have  $n/2$  switches by the pictorial definition of the network. Each of the subnetworks  $P_A$  and  $P_B$  has half the number of inputs that the whole network does. Therefore, following the definition, each has only half as many switches in its first and last columns. There are, however, two such subnetworks so the total number of switches in the second column and the next to last column remains  $n/2$ . The same reasoning holds through the remainder of the recursion—4 subnetworks with  $n/4$  inputs and so on down to a single column of  $n/2$  permuters of two inputs (switches). There are  $2\lg n - 1$  columns where  $\lg n$  stands for  $\log_2 n$ . Each invocation of the definition adds two columns except for the last which adds only one column.

### 1.3 The Connection Machine System

In this section we briefly introduce the characteristics of the CM that are of interest in the context of the algorithms developed and/or implemented in the thesis. Further details on the architecture of the CM can be found in Hillis [57] and system documentation.

The basic component of the CM is an integrated circuit with sixteen single-bit processing elements (PE) and a router that handles general communication. A fully configured CM-2 includes 4,096 chips for a total of 65,536 PEs and 2048 floating point units (each floating point unit is shared by 32 processors across 2 chips). The 4,096 chips are wired together as a 12-dimensional hypercube. That is, assuming the chips are assigned addresses from 0 to 4,095, two chips are connected by a wire if and only if the binary representation of their addresses differ in exactly one bit. For our purposes, the Connection Machine is best viewed as an 11-dimensional complete hypercube with a floating point unit at each node. In this view of the Connection Machine, each unit has 64K 32-bit words of local memory. Figure 1-3 shows a block diagram of the Connection Machine.

The Connection Machine is attached to a front end sequential processor which holds the programs and executes any sequential code. Operations by the PEs are under the control of a microcontroller that broadcasts instructions simultaneously to all the elements for execution. A flag register at every PE allows for no-operations; i.e. an instruction received from the microcontroller is executed if the flag is set 1 and ignored otherwise. The concept of virtual

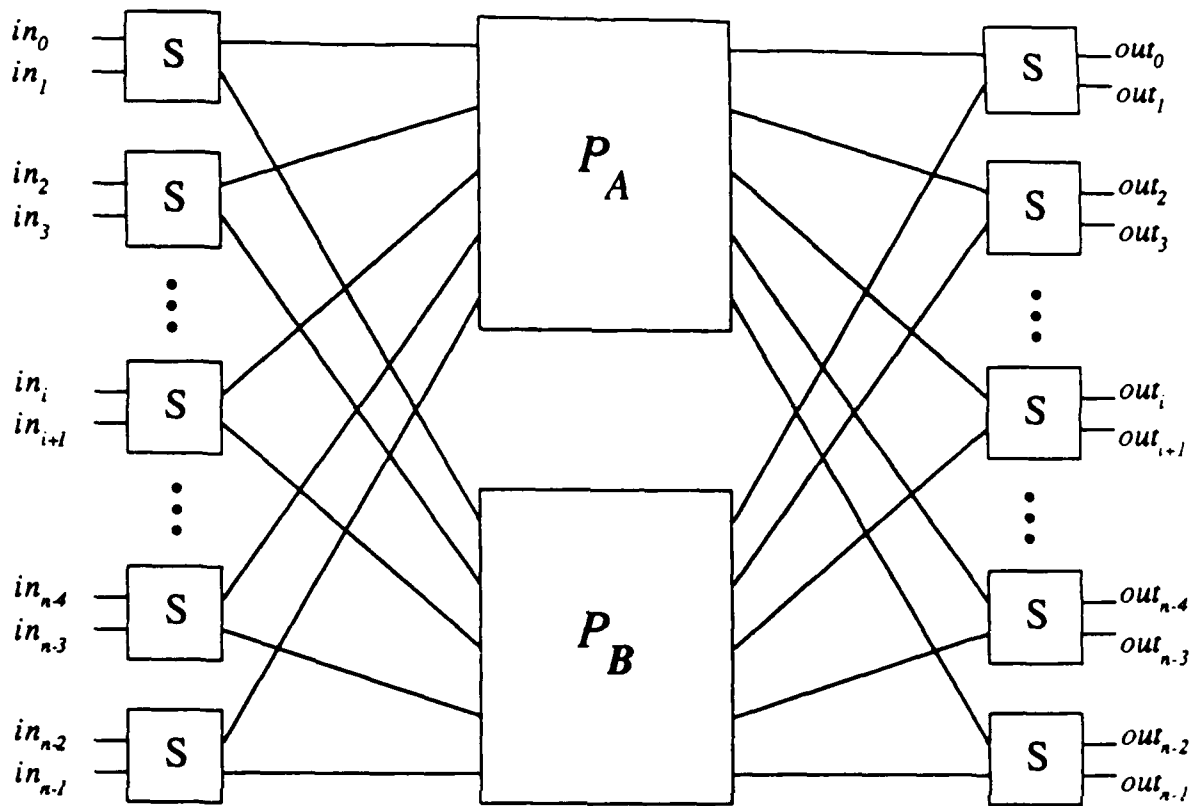


Figure 1-2: The Benes network with  $n$  inputs. The boxes with the "S" are primitive switches. The boxes labeled  $P_A$  and  $P_B$  are Benes networks of  $n/2$  inputs constructed recursively.

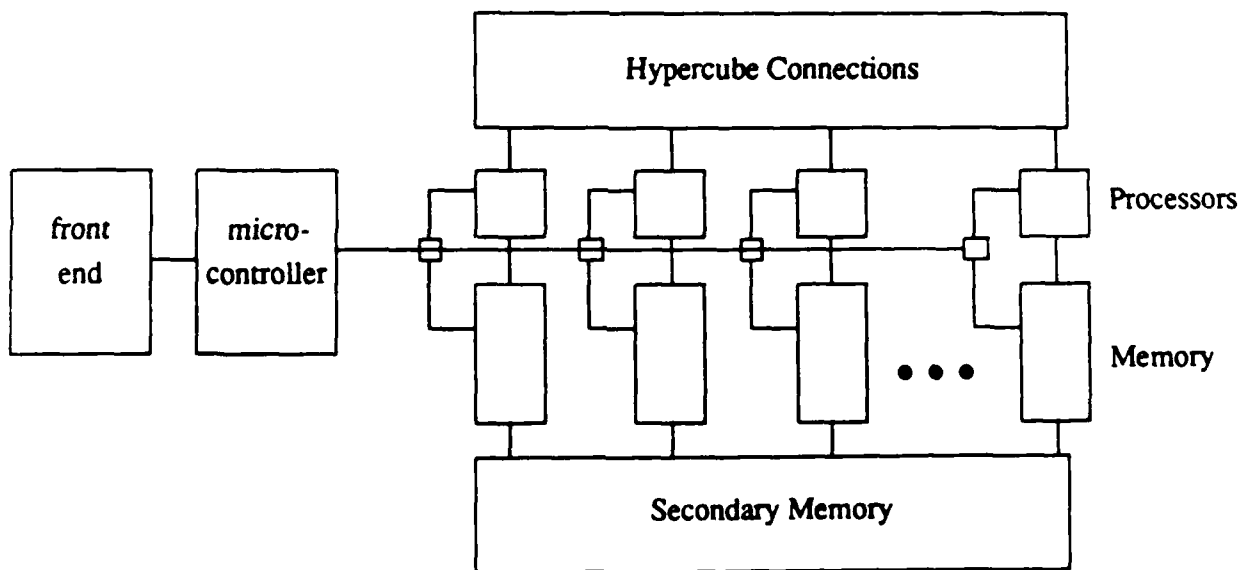


Figure 1-3: A block diagram of the Connection Machine.

*processors* (VP) allows an application to use more PEs than the number of physical elements available. A VP is specified by slicing the memory of a physical processor and then allowing the physical PE to loop over all the slices. Such a looping is executed in linear time.

Using standard Gray coding, the processors of the CM can be configured as an  $n$ -dimensional grid such that two processors that are neighbors in the grid are connected by a hypercube wire. Thus communication within a grid — called NEWS communication for North, South, East, and West — is particularly simple since the required data paths can be precomputed and the general router bypassed. The CM software system provides general  $n$ -dimensional NEWS communication, all transparent to the user, but our applications require only 2-dimensional grids. The configuration of the NEWS grid (length of the axes) is determined by the user at the time the CM is cold-booted and the software system then automatically provides the mapping from grid coordinates to CM processors. General intercommunication and dynamic reconfiguration is performed by the much more powerful *router* communication system. The router uses the hypercube wires to deliver a message from each VP to any other VP. Because the communication pattern is arbitrary and can contain collisions, the time needed to deliver a given set of messages varies.

Two models of Connection Machines are currently available: the CM-1 and the CM-2. From a high-level language programming point of view the two systems are identical and programs written for the CM-1 can migrate without modifications to the CM-2. There are, however, some hardware differences that significantly improve the performance of the CM. The CM-2 is equipped with floating point accelerators, each processor has more memory (initially 8K bytes and now larger), and extra hardware assists in the combination of messages and delivery of messages to processor memory. In particular, the CM-2 has indirect addressing which allows different processors to access different locations within their memory at the same time. Thus the CM-2 deviates somewhat from the strict data parallel model.

The programming languages currently available for the CM are \*LISP and C\* — extensions of LISP and C respectively — and Fortran8x. These languages allow control of the PEs and the specification of parallel variables. Our applications are programmed in \*Lisp, Paris (the Connection Machine assembly language) and/or microcode. Parallel \*Lisp primitives that are relevant to our implementation are *Scans*, *\*PSet*, and *Global* operations. Blelloch [16] provides details on these, and other, primitives of the CM.

Scan is also known in the literature as parallel prefix. Given an associative, binary operator  $\otimes$ , the  $\otimes$ -scan primitive takes a sequence  $\langle x_0, x_1, \dots, x_n \rangle$  and produces another sequence  $\langle y_0, y_1, \dots, y_n \rangle$  such that  $y_i = x_0 \otimes x_1 \otimes \dots \otimes x_i$ . On the Connection Machine, for example, *Plus-Scan* takes as an argument a parallel variable (i.e. a vector with each element residing in the memory of a different PE) and returns to PE  $i$  the value of the parallel variable summed over  $j = 0, \dots, i$ . *Copy-Scan* takes as an argument a parallel variable and copies its first value to the memory of all PEs. Options specified in the scan call allow the scan to apply only to preceding processors (eg. sum from  $j = 0 \dots i - 1$ ) or to perform the scan in reverse. Another variation of the scan primitives allows their operation within *segments* of a parallel variable or PEs. These primitives are denoted, for example, as *Segmented-Plus-Scan* and *Segmented-Copy-Scan*. They take as arguments a parallel variable and a set of segment bits which specify how to partition the set of processors into contiguous segments. Segment bits have a 1 at the starting location of a new segment and a 0 elsewhere. A *Segmented-Scan* operation re-starts at the beginning of every segment. When processors are configured as a NEWS grid, scans

within rows or columns are special cases of segmented scans called *grid-scans*. Because the communication pattern needed to perform the calculations can be predetermined, scans use the hypercube wires directly, bypassing the router, and generally require less time than a single general route.

The *\*PSet* primitive takes as arguments two parallel variables and a set of memory addresses and copies the first argument to the second at the PEs given by the memory addresses. Because the communication pattern is arbitrary, this primitive uses the router.

*Global* operations on parallel variables return a single value to the front end processor. For example, we can calculate the sum or the maximum of an integer parallel variable or the  $\wedge$  (and) of a boolean parallel variable. The front end can also broadcast values to all processors in parallel.

Because there is a wire to each processor from the front end used for broadcasting, the Connection Machine has a *wired OR* facility. That is, the front end can detect if some processor is pulling down the wire (though it cannot tell which one). This wired-OR facility leads to fast global integer maximum calculations (called *\*max*). The integers are processed bit serially starting with the most significant bit. All processors containing a value with most significant bit equal to 1 pull down the wire. If the wire is pulled, then all processors with a first bit of 0 drop out of the process. All remaining processors proceed to the next bit and so on. The front end processor can thus read out the maximum bit-serially starting with the most significant bit.

## Chapter 2

# Parallel Graph Contraction

### 2.1 Introduction

The parallel tree contraction technique of Miller and Reif [88] has proved to be a valuable tool in many parallel graph algorithms. This chapter provides a similar contraction technique that applies not only to trees, but also to general graphs. It also provides a second, potentially faster contraction technique for bounded-degree graphs (and general graphs when an embedding is known). Portions of this chapter represent joint work with Charles Leiserson.

A *contraction step* of an undirected graph  $G = (V, E)$  with respect to an edge  $(u, v) \in E$  is the operation that replaces  $u$  and  $v$  by a new vertex  $w$  which is adjacent to all those vertices to which  $u$  and  $v$  were adjacent. If vertices  $u$  and  $v$  have an adjacent vertex  $x$  in common, the contraction step produces only one edge between  $w$  and  $x$  rather than two. Thus, the graph that results from a contraction step is not a multigraph. A *parallel paired contraction step* is a vertex-independent sequence of contraction steps. A *parallel multi-contraction step* is a sequence of contraction steps with respect to an acyclic set of edges. We refer to the sequence as simply a parallel contraction step whenever the type of contraction is clear from context. A parallel paired contraction step of any bounded-degree graph can be implemented in constant parallel time. A (*parallel*) *contraction* of a graph is the process by which a connected graph is reduced to a single node by iterated (parallel) contraction steps.

The first contraction-like algorithm was introduced by Reif when he used a "random mate" technique in his  $O(\lg n)$ -time linear-processor connected components algorithm for the CRCW PRAM model [103]. Miller and Reif [88] show how any tree can be contracted in  $O(\lg n)$  time using  $O(n)$  processors in the CRCW PRAM model using randomization. (They also show how to reduce the number of processors to  $O(n/\lg n)$ .) They give an  $O(\lg n)$ -time algorithm for tree contraction that is deterministic, but the deterministic algorithm does not perform contraction in the strict sense described above. Tree contraction can be made to run in randomized  $O(\lg n)$  time and in deterministic  $O(\lg n \lg^* n)$  time using  $O(n)$  processors in the EREW and DRAM models, as was shown in [78].

In this chapter, we use the technique of parallel contraction in a more general setting. We show that a remarkably simple contraction algorithm that uses  $(n + e)/\lg n$  processors reduces a connected  $n$ -node  $e$ -edge graph to a single node in  $O(\lg^2 n)$  steps and a second simple algorithm which uses the first as a subroutine reduces a connected bounded-degree graph to a single node using  $n/\lg n$  processors in  $O(\lg n + \lg^2 \gamma)$  steps, where  $\gamma$  is the maximum genus

of any connected component of graph  $G$ . We do not need to know  $\gamma$  to achieve a running time that is a function of  $\gamma$ . Neither algorithm requires an embedding of the graph. If we do have an embedding, however, the bounds of the second algorithm are applicable to any graph. The second algorithm immediately yields an asymptotically efficient solution to the problem of region labeling in vision systems, as well as to solutions of other planar graph problems.

The genus of a graph is the sum of the genus of the connected components. That is, if the  $i$ th connected component has genus  $\gamma_i$ , then the genus of the graph is  $\gamma_G = \sum_i \gamma_i$ . The running time of our second algorithm, however, depends only upon the *component genus* which is defined to be the maximum genus of any connected component, (i. e.  $\gamma = \max_i \gamma_i$ ). For example, a graph with  $\Theta(n)$  toriodal components has a genus that is  $\Theta(n)$  but a component genus  $\gamma = 1$ . Henceforth, when we refer to the genus of a graph  $G$ , we will use the notation  $\gamma_G$  when we mean the true genus and we will use the notation  $\gamma$  when we mean the component genus of graph  $G$ .

The remainder of the chapter is organized as follows. Section 2.2 describes the data structure used to represent general graphs, motivates the decision, and illustrates how to perform a contraction step on the data structure. Section 2.3 presents the contraction algorithm for general graphs, argues its correctness, and analyses its running time. Section 2.4 presents the contraction algorithm for bounded-degree graphs and Section 2.5 analyzes its running time using a "missing edge" lemma which is proved in section 2.6. Section 2.7 shows how graph contraction can be applied to various graph problems, including the region-labeling problem. Section 2.8 offers some concluding remarks.

## 2.2 The Graph Data Structure

In this section we describe the data structure used in the general contraction algorithm, show how to perform a contraction step, and illustrate why we chose this structure.

Figure 2-1 illustrates the data structure we use in our algorithms. We represent each vertex of degree  $d$  by a doubly-linked ring of  $d$  *real processors* alternating with  $d$  *dummy processors*. The edges between real processors and dummy processors, represented by heavy lines in figure 2-1, are called *vertex edges* and edges between vertex rings are called *graph edges*. Each processor in a vertex ring knows the ID of the vertex, defined to be the maximum identifier of any processor in the vertex ring.

Figure 2-2 illustrates how to contract an edge  $(u, v)$  by using the dummy processors to merge the two vertex rings and splice out the real processors corresponding to the contracting edge. The dummy edges allow us to simultaneously contract any acyclic set of edges, including trees of arbitrary depth, in constant time. We then pay an extra  $O(\lg n)$  time to remove the extra dummy vertices.

The dummy processors are needed for efficient implementation of a parallel multi-contraction step. Suppose that we did not use dummy processors, but instead formed vertex rings of real processors only as shown in figure 2-3 for the graph of figure 2-1. If we wish to contract simultaneously two edges associated with neighbors in a vertex ring, we find we can no longer perform simple, local updates. For example, in figure 2-3 if we wish to contract the double hatched edges  $(b, d)$  and  $(b, e)$  simultaneously, and we use the simple contraction strategy of figure 2-2, we find that when processor  $b_2$  splices itself out, it uses processor  $b_3$  which is simultaneously trying to splice itself out of the ring by using processor  $b_2$ . The resulting data structure has a disconnected ring that includes processors  $b_2$  and  $b_3$  which should have been eliminated.

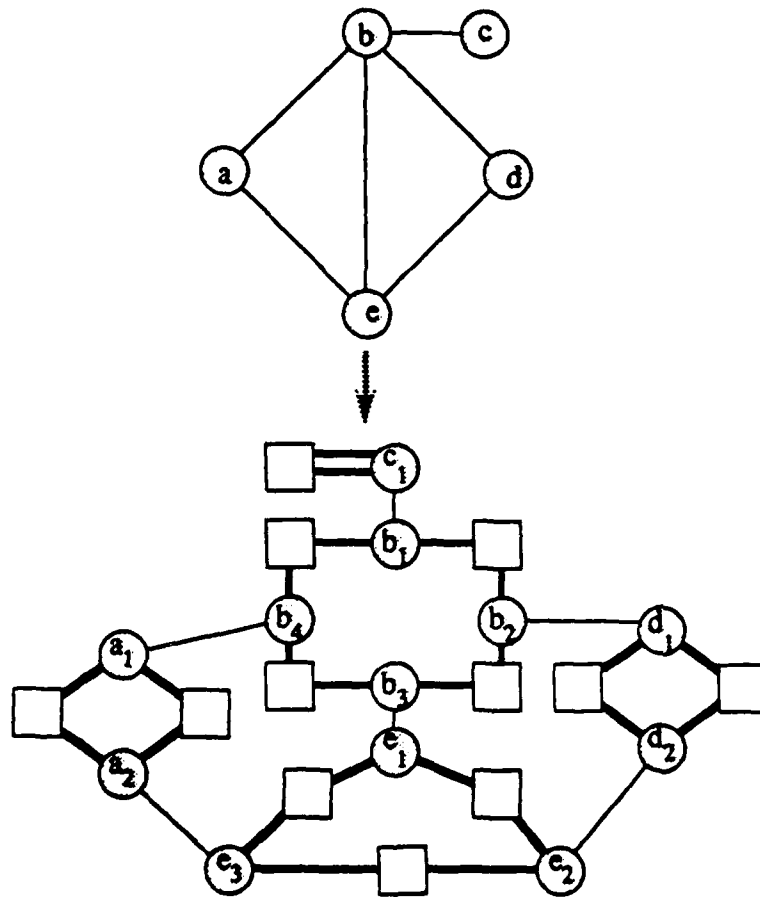


Figure 2-1: We represent graphs using a ring of processors for each vertex. Two (round) *real* processors are separated by a (square) *dummy* processor in the list. Vertex edges, drawn as heavy lines, go between processors in the same vertex ring, while graph edges (thin lines) go between processors in different vertex rings.

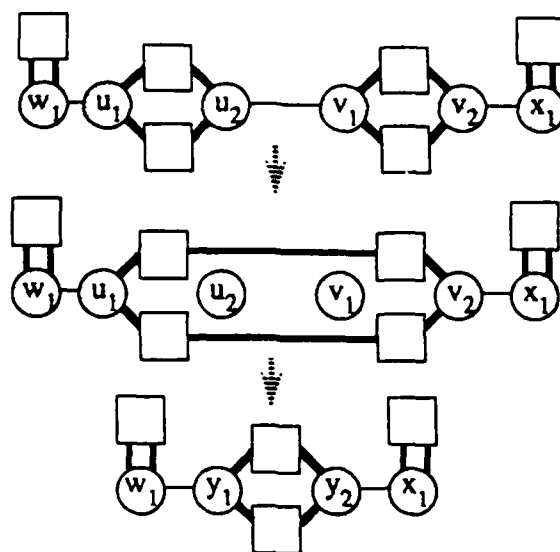


Figure 2-2: To contract a graph edge  $(u, v)$ , we use the neighboring dummy processors on each side to merge the vertex rings and splice out the real processors. We later clean up the extra dummy processors. Dummy processors allow contraction of trees of arbitrary depth.

To perform this update correctly for trees of arbitrary depth would require searches of arbitrary length across an arbitrary number of vertex rings. Alternatively, we could incorporate a strategy which would not allow edges which are neighbors in a vertex ring to contract simultaneously. This complicates the algorithm by requiring some form of symmetry-breaking within each vertex ring to decide which processors can participate in a contraction and edges could contract only if both processors associated with that graph edge were allowed to contract. We would then in general require more iterations to complete a graph contraction because fewer edges are allowed to contract during each iteration. By using dummy processors, any set of acyclic edges can contract simultaneously by local splicing. Then extra dummy processors are removed by pointer jumping strictly within vertex rings.

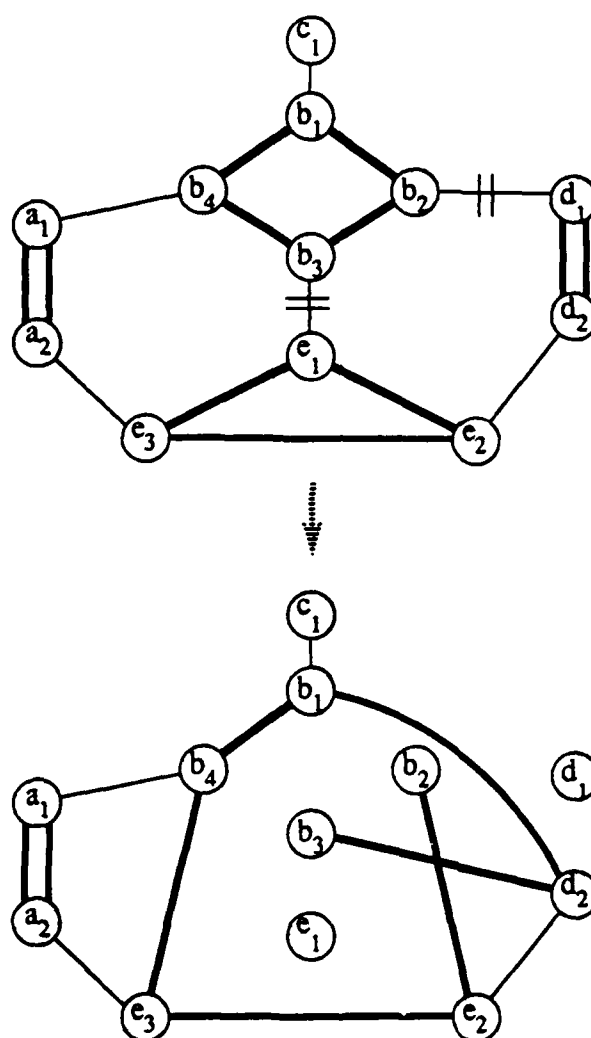
## 2.3 General Algorithm

This section presents an  $O(n + e)$ -processor contraction algorithm for general graphs, argues its correctness and analyzes its running time. We give a strategy for determining a set of acyclic edges to contract in a parallel multi-contraction step. We then discuss implementation issues such as avoiding sorting, maintaining linear space, and reducing the processor count to  $O((n + e)/\lg n)$ .

The algorithm for contracting graphs is simple. It uses a subroutine  $\text{MERGE}(u, v)$  which performs the edge contraction described above for edge  $(u, v)$ .

### Algorithm CONTRACT

- 1 While  $\exists$  a vertex with degree  $> 0$  do
- 2   Each vertex  $u$  with degree  $> 0$ , do in parallel



**Figure 2-3: If we did not use dummy processors, the simple splicing strategy would fail if neighbors in the vertex ring tried to contract simultaneously.**

- 3     Let  $v$  be the neighbor of highest ID
- 4     MERGE( $u, v$ )
- 5     Remove extra dummy processors
- 6     Remove multiple edges to adjacent vertices
- 7     Propagate new ID.

We now argue the correctness of Algorithm CONTRACT. The only subtle point is that simultaneous contraction of a set of edges that form a tree results in a single vertex ring. If we were to contract edges that form a cycle, then we may end up with multiple rings, thus severing a connected component. Because each vertex chooses to merge with its neighbor of maximum ID, the edges chosen to contract in an iteration cannot form a cycle.

We now analyze the running time of algorithm CONTRACT. Each iteration of the loop requires  $O(\lg n)$  time. The loop termination check in line 1 can be done in  $O(\lg n)$  time. Each processor can determine in constant time whether the vertex ring to which it belongs has degree  $> 0$ . We can remove multiple edges in  $O(\lg n)$  time by first sorting the processors in a vertex ring with respect to the ID of the neighbor across the graph edge and then pointer jumping. Similarly, propagating the highest ID of any neighbor to all processors in a ring, removing extra dummy processors, and propagating the new vertex ID can all be done in  $O(\lg n)$  time using pointer jumping. If we are careful, all pointer jumping can be done without using concurrent reads. The MERGE operation requires only constant time. Thus each iteration requires  $O(\lg n)$  time. The algorithm terminates in  $O(\lg n)$  iterations because each vertex merges on every iteration, at least halving the number of vertices.

We can avoid sorting if we allow multiple edges to remain after contraction and instead use pointer jumping to remove the self edges that appear in later iterations. If we choose this option, then in line 3 we must also break ties among all edges to vertex  $v$ . Using the idea of "spares" from [78], the algorithm can be made to use only linear space.

To reduce the processor count to  $(n + e)/\lg n$ , each processor simulates  $k$  processors per iteration (initially  $k = \lg n$ ). After each iteration of the contraction algorithm, we balance the work among the processors by enumerating the remaining nodes in  $O(\lg n)$  time via parallel prefix and compacting memory in  $O(k)$  time. Thus each iteration requires  $O(k \lg n)$  time. The parameter  $k$  is initially  $\lg n$  and decreases exponentially with each iteration, so overall the contraction algorithm runs in  $O(\lg^2 n)$  time using  $(n + e)/\lg n$  processors. The balancing steps can be done without concurrent memory access, so these bounds hold for the EREW PRAM.

## 2.4 Bounded-Degree Algorithm

In this section we present a linear-processor contraction algorithm for bounded-degree graphs. We give a randomized strategy for determining a set of vertex-disjoint edges to contract on a given parallel contraction step. We discuss simplifications for the case where the graph is known to be planar or of low genus. We discuss implementation issues including maintaining linear space, using this algorithm for arbitrary graphs provided an embedding is known, making the algorithm deterministic, and reducing the processor count. Finally we mention some of the parallel models for which the analysis of section 2.5 holds.

The contraction algorithm for bounded-degree graphs is as follows. We assume that no vertex has degree greater than 40. The number  $j$  in line 1 is a constant chosen such that algorithm BOUNDED-CONTRACT terminates in  $O(\lg n + \lg^2 \gamma)$  time with high probability. The constant will be discussed during the analysis of the algorithm in section 2.5.

*Algorithm BOUNDED-CONTRACT*

```

1 Repeat  $j \lg n$  times
2   for each vertex  $u$  do in parallel
3     Randomly choose an adjacent vertex  $v$  such that MERGE( $u, v$ )
       would produce a vertex of degree at most 40
4   for all vertices  $u$  and  $v$  that choose each other, do MERGE( $u, v$ )
5   for each vertex  $u$  do
       remove multiple edges to adjacent vertices
6 CONTRACT() go to general contraction algorithm

```

The algorithm can be broken down into two phases: phase 1 in lines 1–5 in which vertices resulting from contraction cannot have degree greater than 40 and phase 2 in line 6 in which vertices resulting from contraction can have arbitrary degree. In section 2.5 we argue that phase 1 runs in  $O(\lg n)$  time and with high probability reduces an  $n$ -vertex genus- $\gamma$  graph to a graph whose largest connected component has  $O(\gamma)$  vertices. By the argument in section 2.3, phase 2 reduces each  $O(\gamma)$ -sized connected component of a graph to a single vertex in  $O(\lg^2 \gamma)$  time. Planar graphs and graphs of constant genus will be reduced to constant-sized graphs during phase 1 of algorithm BOUNDED-CONTRACT with high probability. If we know that a graph is of constant genus, we can simplify the algorithm to repeated application of lines 2–5 only, until the graph is contracted. The parameter 40 in line 3 must be replaced by  $c(\gamma)$ , a constant depending upon the genus  $\gamma$ . In particular,  $c(\gamma) = 40$  suffices for planar graphs. We then check for termination (all vertices degree 0) every  $\lg n$  parallel contraction steps. This simplified algorithm will contract an  $n$ -vertex bounded-genus graph to a single vertex in  $O(\lg n)$  time with high probability. Furthermore, because the degree of each vertex remains bounded throughout the algorithm in this case, we can simply assign one processor to each vertex.

The algorithm can be made to work in  $O(\lg n + \lg^2 \gamma)$  time for any graph whose maximum degree is a constant greater than 40 by replacing the number 40 in line 3 with the maximum degree. Moreover, the algorithm can be made to work in  $O(\lg n + \lg^2 \gamma)$  time for any graph, provided an embedding for the graph is known, since we can transform any general graph of genus  $\gamma$  into a degree-3 genus- $\gamma$  graph by replacing each vertex by a ring of degree-3 vertices in the appropriate cyclic order. Again, using the idea of “spares” from [78], the algorithm can be made to use only linear space.

The contraction algorithm can be modified to run in deterministic  $O(\lg n \lg^* n + \lg^2 \gamma)$  time by using the  $O(\lg^* n)$ -time algorithm of Goldberg, Plotkin, and Shannon for 3-coloring rooted trees [48] to guarantee that a constant fraction of vertices merge on each iteration. Randomness is used only in the choice of vertex pairings in phase 1. In the deterministic version of the algorithm, each vertex  $u$  chooses the vertex  $v$  with smallest identifier such that MERGE( $u, v$ ) would produce a vertex of degree at most  $d_{\max}$ , the maximum degree of any vertex. As before, edges chosen by both vertices adjacent to it are automatically selected to contract. Consider the graph induced by the edges chosen by exactly one adjacent vertex. If the edges are directed such

that the vertex choosing the edge is at the tail, then this graph is a bounded-degree directed forest with edges directed from child to parent. We can then use the Goldberg-Plotkin-Shannon  $O(\lg^* n)$ -time algorithm for coloring bounded-degree trees to color the forest using a constant number of colors [48]. Then, we sequence through the colors allowing edges chosen by vertices of the current color to contract, provided the vertex on the other end of the edge has not already participated in a contraction during this parallel contraction step. In this scheme, we always have a constant fraction of the vertices merging.

We can reduce the processor requirement of phase 1 to  $n/\lg n$  by using the techniques of Gazit and Reif [46] who in turn use the load balancing techniques of Cole and Vishkin [26]. Shannon [110] gives a scheduling algorithm that converts the deterministic  $O(\lg n \lg^* n)$ -time  $n$ -processor algorithm for graphs of bounded genus to an optimal  $O(\lg n \lg^* n)$ -time  $n/(\lg n \lg^* n)$ -processor algorithm.

Algorithm BOUNDED-CONTRACT is robust in that it can be implemented in several of the most restrictive parallel models. By careful attention to the data structures used to implement adjacency lists, it is possible to guarantee that no concurrent reading or writing occurs, and thus, the performance bounds apply in the EREW PRAM model. Since each processor is responsible for a single edge (or vertex) of the graph, it is naturally a “data-parallel” algorithm in the sense of [58]. Finally, the simplified version of the algorithm which uses only phase 1 is “conservative” in the sense of [78], and thus runs in  $O(\lg n + \lg^2 \gamma)$  steps in the DRAM model.

## 2.5 Analysis of the Contraction Algorithm

In this section we analyze the contraction algorithm of section 2.4. We require two lemmas. Lemma 1 (the Missing-Edge Lemma) provides an upper bound on the number of edges in a graph based on Euler’s formula and the number of degree-three vertices that are ineligible for contraction. The missing-edge lemma is actually proved in section 2.6. We use a pigeonholing argument to show in Lemma 2 that at each parallel contraction step of phase 1, a constant fraction of the vertices are adjacent to at least one edge that can be contracted — This lemma is essentially the same as one proved independently by Boyar and Karloff [20] in the context of coloring planar graphs, but our lemma is more general and our proof differs somewhat. The final result of the section proves that with high probability, the running time of Algorithm BOUNDED-CONTRACT is  $O(\lg n + \lg^2 \gamma)$  for bounded-degree graphs.

We analyze Algorithm BOUNDED-CONTRACT using a constant  $d_{\max} \geq 40$  in place of 40 in line 3 of the contraction algorithm. Using a symbolic value allows us to see in the analysis why we choose degree 40 as the maximum degree in the algorithm and to see how the contraction algorithm generalizes to bounded-degree graphs in general. In fact, a choice of  $d_{\max} = 3$  suffices for Algorithm BOUNDED-CONTRACT to contract binary trees in randomized  $O(\lg n)$  time.

We first present some definitions. Let  $G = (V, E)$  be a graph with degree at most  $d_{\max} \geq 40$ . We call an edge  $(u, v) \in E$  *eligible* if MERGE( $u, v$ ) would produce a vertex  $w$  of degree at most  $d_{\max}$ , as in line 3 of Algorithm BOUNDED-CONTRACT. Typically, the degree of  $w$  is  $\deg(w) = \deg(u) + \deg(v) - 2$ , but it is less whenever the adjacency lists of  $u$  and  $v$  have vertices in common because a contraction step removes multiple edges. (In fact, after the parallel contraction step implemented by Algorithm BOUNDED-CONTRACT, the degree of vertex  $w$  may be even smaller, since lines 5 and 6 remove additional multiple edges caused by other simultaneous contraction steps.) We define a vertex  $u \in V$  to be *good* if it is incident on at

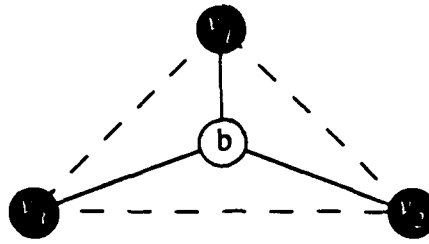


Figure 2-4: Degree-three vertex  $b$  is bad only if it is adjacent to three independent degree- $d_{\max}$  vertices, shown in black. The dashed edges must be missing from the graph if vertex  $b$  is to be bad even though their inclusion does not increase the genus of the graph. Thus, graphs containing bad degree-three nodes are sparser than required by Euler's formula alone.

least one eligible edge, and *bad* otherwise. A vertex of degree 1 or 2 is automatically good. A vertex of degree 3 is good unless it is incident on three degree- $d_{\max}$  vertices  $v_1$ ,  $v_2$ , and  $v_3$  that are independent (no edge between any pair) as shown in figure 2-4. Consequently, a bad degree-3 vertex  $b$  causes the edges  $(v_1, v_2)$ ,  $(v_2, v_3)$ , and  $(v_1, v_3)$  to be *missing* from the graph, even though their inclusion would not increase the genus of the graph.

The next lemma uses the notion of missing edges to show that a graph with bad degree-3 vertices is sparser than required by Euler's formula alone. It is proved in section 2.6.

**Lemma 1** (*Missing-Edge Lemma*) A graph  $G = (V, E)$  of genus  $\gamma_G$  with  $b_3 \geq 0$  bad degree-3 vertices has at most  $3|V| - b_3 + 10\gamma_G$  edges.

The next lemma uses the Missing-Edge Lemma and pigeonholing to show that during each parallel paired contraction step in phase 1 of algorithm BOUNDED-CONTRACT, a constant fraction of the vertices are eligible to contract.

**Lemma 2** Any genus- $\gamma_G$  graph  $G = (V, E)$  with degree at most  $d_{\max} \geq 40$  and  $|V| \geq 40\gamma_G$  has at least  $\frac{1}{12}|V|$  good vertices.

*Proof.* We begin the proof by defining the following sets:

- $V_{\text{good}}$  is the set of good vertices,
- $V_{\text{bad3}}$  is the set of bad degree-3 vertices,
- $V_{\text{bad456}}$  is the set of bad vertices of degrees 4, 5, or 6,
- $V_{\text{high}}$  is the set of vertices of degrees  $d_{\max} - i$ , for  $i = 0, 1, 2$  or 3,
- $V_{\text{rest}} = V - (V_{\text{good}} \cup V_{\text{bad3}} \cup V_{\text{bad456}} \cup V_{\text{high}})$ .

having cardinalities  $g$ ,  $b_3$ ,  $b_{456}$ ,  $h$ , and  $b_{\text{rest}}$  respectively.

We determine a lower bound on the number  $g$  of good vertices using three constraints. The first constraint is a lower bound on the number  $e$  of edges in  $E$ , which is also half the sum of the degrees of all vertices in  $V$ . For each of the sets defined above, we underestimate the sum of the degrees of the vertices in the set to yield

$$\begin{aligned} e &\geq \frac{1}{2}(g + 3b_3 + 4b_{456} + (d_{\max} - 3)h + 7b_{\text{rest}}) \\ &= \frac{1}{2}(g + 3b_3 + 4b_{456} + (d_{\max} - 3)h + 7(n - g - b_3 - b_{456} - h)). \end{aligned}$$

We can use a similar technique to determine a lower bound on the number of high-degree vertices. The number of edges leaving the set  $V_{\text{high}}$  is at most  $d_{\text{max}}h$ , which must be at least the number leaving the sets  $V_{\text{bad3}}$  and  $V_{\text{bad456}}$  since each of the vertices in these sets is adjacent only to vertices in  $V_{\text{high}}$ . Hence, we obtain

$$h \geq \frac{3b_3 + 4b_{456}}{d_{\text{max}}}. \quad (2.1)$$

Finally, from the Missing-Edge Lemma we have

$$e < 3n - b_3 + 10\gamma_G. \quad (2.2)$$

We use these three constraints to obtain the desired lower bound on the number  $g$  of good vertices. Combining Inequalities (2.1) and (2.2) and solving for  $g$ , we obtain

$$g > \frac{1}{6}((d_{\text{max}} - 10)h - 2b_3 - 3b_{456} + n - 20\gamma_G).$$

Substituting for  $h$  in this inequality using Inequality (2.1) yields

$$\begin{aligned} g &> \frac{1}{6} \left[ \left( \frac{d_{\text{max}} - 10}{d_{\text{max}}} \right) (3b_3 + 4b_{456}) - 2b_3 - 3b_{456} + n - 20\gamma_G \right] \\ &= \frac{1}{6} \left[ \left( \frac{d_{\text{max}} - 30}{d_{\text{max}}} \right) b_3 + \left( \frac{d_{\text{max}} - 40}{d_{\text{max}}} \right) b_{456} + n - 20\gamma_G \right] \\ &\geq \frac{1}{6}(n - 20\gamma_G), \end{aligned}$$

if  $d_{\text{max}} \geq 40$  as assumed. Since we also assume that  $n \geq 40\gamma_G$ , we have that

$$\begin{aligned} g &> \frac{1}{6}(n - 20\gamma_G) \\ &\geq \frac{1}{6} \left( n - \frac{n}{2} \right) \\ &= \frac{1}{12}n. \end{aligned}$$

which completes the proof. ■

The proof that Algorithm BOUNDED-CONTRACT runs in  $O(\lg n + \lg^2 \gamma)$  time uses two technical lemmas from [78]. The first lemma provides a lower bound on the probability that a random variable with finite upper bound will exceed a given value.

**Lemma 3** *Let  $X \leq b$  be a discrete random variable with expected value  $\mu$ . For  $w < b$ , we have*

$$\Pr(X \geq w) \geq \frac{\mu - w}{b - w}.$$

The second lemma provides a "Chernoff"-type bound [24] on the tail of a binomial distribution. Consider a set of  $t$  independent Bernoulli trials, each with a probability  $p$  of success. The next lemma bounds the probability  $B(s, t, p)$  that fewer than  $s$  successes occur in  $t$  trials when  $t > 2s$  and  $p < 1/2$ .

**Lemma 4** For  $t > 2s$  and  $p < 1/2$ , we have

$$B(s, t, p) \leq \left( \frac{1-p}{1-2p} \right) \left( (1-p)^t \right) \left( \frac{et}{s} \right)^s.$$

We now analyze the behavior of phase 1. Phase 1 runs in  $O(\lg n)$  time. Given a graph  $G$  with maximum vertex degree at most 40, Algorithm BOUNDED-CONTRACT does not allow the degree of any vertex to exceed 40 during phase 1 (the parallel contraction steps in lines 2-5 of Algorithm BOUNDED-CONTRACT). Therefore, each such step can be performed in constant time, since contraction and removal of multiple edges involves only communicating around constant-length vertex rings. Since we execute  $j \lg n$  parallel contraction steps for a constant  $j$ , phase 1 is over in  $O(\lg n)$  time.

We now use Lemmas 2, 3, and 4 to show that for suitable choice of constant  $j$ , with high probability phase 1 reduces an  $n$ -vertex genus- $\gamma$  graph to a graph with  $O(\gamma)$  vertices in its largest connected component. We need only show that with high probability,  $O(\lg n)$  parallel contraction steps suffice to contract each connected component to size  $O(\gamma)$ . From lemma 2, we have that at least  $1/12$  of the vertices are good provided that there are at least  $40\gamma_G$  vertices. The lemma applies independently to each connected component. For component  $i$ , contraction steps have a  $o(1)$  probability of success only after the component size has been reduced to  $O(\gamma_i)$  where  $\gamma_i$  is the genus of component  $i$ .

**Theorem 5** After  $O(k \lg n)$  parallel paired contraction steps, any connected, degree-40, graph of genus  $\gamma$  has contracted to a graph with  $O(\gamma)$  vertices with probability  $1 - O(1/n^k)$  for any constant  $k$ .

*Proof.* During each parallel contraction step, each vertex chooses an edge randomly out of all adjacent edges eligible for contraction. Since in phase 1 no vertex degree exceeds 40 throughout the contraction, on each iteration every good vertex  $u$  has at least a  $1/40$  probability of merging, since  $1/40$  is a lower bound on the probability that the edge  $(u, v)$  that vertex  $u$  chooses is also chosen by the vertex  $v$  at the other end. Lemma 2 shows that a bounded-degree genus- $\gamma$  graph of size  $40\gamma$  will have at least  $1/12$  of its vertices be good during each iteration of phase 1. Therefore, we expect that at least  $n/480$  of the vertices of an  $n$ -vertex bounded-degree graph will contract on each iteration of phase 1 provided that  $n$  is sufficiently large compared to the genus of the graph.

We will call an iteration *successful* if at least  $n/960$  vertices pair on that iteration. Let  $P$  be the number of vertices pairing. Then by Lemma 3 we have the following lower bound on the probability that an iteration is successful:

$$\Pr(P \geq n/480) \geq \frac{\frac{n}{480} - \frac{n}{960}}{n - \frac{n}{960}} = \frac{1}{959}.$$

During a successful iteration, the number of vertices in the graph is reduced by a factor of at least  $1/\beta \equiv 959/960$ . Therefore we require at most  $\log_\beta n$  successful iterations to reduce the graph to size  $O(\gamma)$ . By Lemma 4, the probability that fewer than  $s = \log_\beta n$  successful iterations occur in  $\alpha k s$  iterations each with a probability  $p = 1/959$  of success is

$$B(\log_\beta n, \alpha k \log_\beta n, 1/959) \leq \frac{958}{957} \left( \left( \frac{958}{959} \right)^{\alpha k} e \alpha k \right)^{\log_\beta n}.$$

We can choose the constant  $\alpha$  sufficiently large so that  $B(\log_\beta n, \alpha k \log_\beta n, 1/959)$  is  $O(1/n^k)$ . ■

We have shown that with high probability, we require only  $O(\lg n)$  parallel contraction steps to reduce connected component  $i$  to size  $O(\gamma_i)$  and hence reduce the maximum connected component to size  $O(\gamma)$ . Therefore phase 1 runs in  $O(\lg n)$  time and with high probability reduces the largest component of the graph to size  $O(\gamma)$ .

Let us now consider the graph at the start of phase 2. Assuming that we are not already done, we have with high probability at most  $O(\gamma)$  nodes in each component. Connected subgraphs contract independently so the asymptotic contraction time during phase 2 is dominated by the time to contract the largest component. If the maximum component has size  $s$ , then by the analysis in section 2.3, phase 2 terminates in  $O(\lg^2 s)$  time. Since we have  $s = O(\gamma)$  with high probability, then phase 2 terminates in  $O(\lg^2 \gamma)$  time with high probability.

Since the time to perform the contraction is simply the sum of the times to perform phase 1 and phase 2, we have that an  $n$ -vertex genus  $\gamma$  graph is contracted by algorithm BOUNDED-CONTRACT in  $O(\lg n + \lg^2 \gamma)$  time with high probability. The probability of algorithm BOUNDED-CONTRACT failing is the probability of failing in phase 1, since phase 2 is deterministic.

The deterministic version guarantees that phase 1 reduces the  $n$ -vertex graph to an  $O(\gamma)$ -vertex graph, where  $\gamma$  is the largest genus of any connected component. Lemma 2 guarantees that in each parallel contraction step, each  $n_i$ -vertex genus- $\gamma_i$  component for which  $n_i = \Omega(\gamma_i)$  has at least  $n_i/12$  good vertices. Of these good vertices, at least  $2/41$  are matched by the symmetry-breaking techniques presented in section 2.4. The worst case is achieved by many vertices with 40 degree-one neighbors. Therefore, each such component is reduced by at least a factor of  $491/492$  after each parallel contraction step. By running phase 1 of algorithm BOUNDED-CONTRACT for  $\lg_{492/491} n$  iterations, we are guaranteed that each component is of size  $O(\gamma_i)$  when we proceed to phase 2. In phase 1, the time required to perform each parallel contraction step is dominated by the  $\Theta(\lg^* n)$  time required to color rooted trees [48]. Therefore phase 1 always terminates in  $\Theta(\lg n \lg^* n)$  time. Because the graph passed to phase 2 is always of size  $O(\gamma)$ , phase 2 always terminates in time  $O(\lg^2 \gamma)$ , and therefore the deterministic contraction algorithm runs in time  $O(\lg n \lg^* n + \lg^2 \gamma)$ .

There is a constant-factor tradeoff between time spent in phase 1 and time spent in phase 2. The constant factor  $1/12$  of good nodes guaranteed by lemma 2 can be replaced by  $1/c$  for any constant  $c > 6$ . The more general version of lemma 2 states that any bounded-degree genus- $\gamma$  graph with at least  $20c\gamma/(c-6)$  nodes has at least  $1/c$  good nodes. If we choose to base our algorithm upon a fraction of  $1/c > 1/12$  good nodes, the constant  $j$  in line 1 of algorithm BOUNDED-CONTRACT decreases. The expected size of the graph passed on to phase 2 increases, however, and therefore we can expect phase 2 to require more contraction steps.

We should comment that although the constants are large in the asymptotic bounds in Theorem 5, the analysis is highly pessimistic. Typically, a vertex has a much greater chance than  $1$  in  $12$  of being good, and if it is good, it typically has more than a  $1$  in  $40$  chance of merging with a neighbor, because the neighbor is unlikely to be incident on  $40$  eligible edges. Consequently, in practice we could reduce the constant  $j$  in line 1 of algorithm BOUNDED-CONTRACT without significantly harming the behavior of the contraction algorithm.

## 2.6 Missing-Edge Lemma

In this section we present the proof of the missing-edge lemma used in the analysis of algorithm BOUNDED - CONTRACT. We begin by defining *cycle splitting* of a connected genus- $\gamma$  graph, a technique that will be used in the inductive proof of the missing-edge lemma. We then prove that performing cycle splitting on a genus- $\gamma$  graph results in a new connected graph of genus strictly less than  $\gamma$  or results in two disjoint graphs whose genus sum to the original genus  $\gamma$ . Finally, the proof of the missing-edge lemma is a double induction on genus and number of bad degree-three vertices. Throughout this section, we assume all graphs are connected. A disconnected graph  $G$  whose genus  $\gamma_G$  is the sum of the genus of its connected components can only be sparser than a connected graph of genus  $\gamma_G$  and hence the disconnected graph cannot have a weaker missing-edge lemma.

Suppose we have a connected graph of genus  $\gamma$  embedded on a surface of genus  $\gamma$  (e.g. a sphere with  $\gamma$  handles). Consider any simple cycle  $\langle v_0, v_1, \dots, v_{l-1} \rangle$  of the graph. As we travel along the cycle from vertex  $v_0$  back to vertex  $v_0$ , we have a well-defined notion of right and left, since the surface upon which the graph is embedded is *orientable* [38]. Thus the vertices adjacent to each vertex  $v_i$  on the cycle can be partitioned into two sets: those that connect to the vertex from the right ( $V_{i,R}$ ) and those that connect to the vertex from the left ( $V_{i,L}$ ). We perform a *cycle splitting* operation as follows. We remove the cycle from the graph, split each vertex  $v_i$  on the cycle into two vertices  $v_{i,R}$  and  $v_{i,L}$  and form them into two cycles  $\langle v_{0,R}, v_{1,R}, \dots, v_{l-1,R} \rangle$  and  $\langle v_{0,L}, v_{1,L}, \dots, v_{l-1,L} \rangle$ . Finally, vertex  $v_{i,R}$  is connected to each vertex in  $V_{i,R}$ , and vertex  $v_{i,L}$  is connected to each vertex in  $V_{i,L}$ . Intuitively, if we view the embedded cycle as having finite thickness, we simply cut it in half. In figure 2-5, cycle  $v_0, \dots, v_3$  is split. The left version is connected to edges entering the cycle from the left and vice versa.

The following lemma will be used to allow application of the induction hypothesis in the proof of the missing-edge lemma.

**Lemma 6** *If we split a cycle  $C = \langle v_0, v_1, \dots, v_{l-1} \rangle$  of a graph  $G$  of genus  $\gamma$  to obtain a new graph  $G'$  of genus  $\gamma'$ , we have either*

1. *graph  $G'$  contains two disjoint components: graph  $G_L$  of genus  $\gamma_L$  and graph  $G_R$  of genus  $\gamma_R$  such that  $\gamma_L + \gamma_R = \gamma$ , or*
2. *graph  $G'$  is connected and  $\gamma' = \gamma - 1$ .*

*Proof.* Suppose we have an embedding of graph  $G$  on an orientable manifold of genus  $\gamma$  such as a sphere with  $\gamma$  handles. By definition of the genus of a graph and of a surface, all the faces of graph  $G$  are simply connected [38]. That is, they can be continuously contracted to a point on the surface. Let  $v$ ,  $e$ , and  $f$  be the number of vertices, edges, and faces respectively of graph  $G$ . Then the *Euler Characteristic* of the surface is defined as  $\chi = v - e + f$ . If we embed any graph on the surface such that all faces are simply connected, the quantity  $v - e + f$  for that graph will always be equal to the Euler characteristic of the surface. Furthermore, by definition, we have that  $\chi = 2 - 2\gamma$  where  $\gamma$  is the genus of the surface (and of any graph which can be embedded on that surface such that each face is simply connected).

Consider now the surface with the embedding of graph  $G$ . We cut the surface along cycle  $C$  and patch the two resulting boundaries with disks as illustrated in figure 2-6. The resulting surface is an orientable manifold that may or may not be connected.

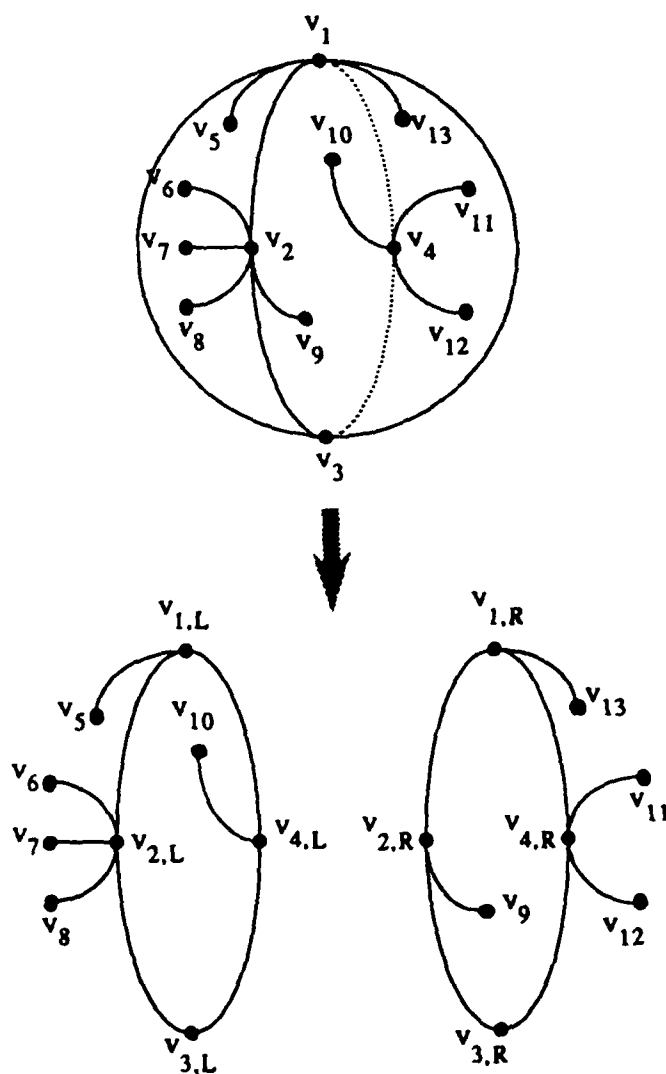


Figure 2-5: Given an embedding of a graph, a cycle can be split into two pieces: the right piece which is connected to nodes on the right as we traverse the cycle, and the left piece which is connected to nodes on the left.

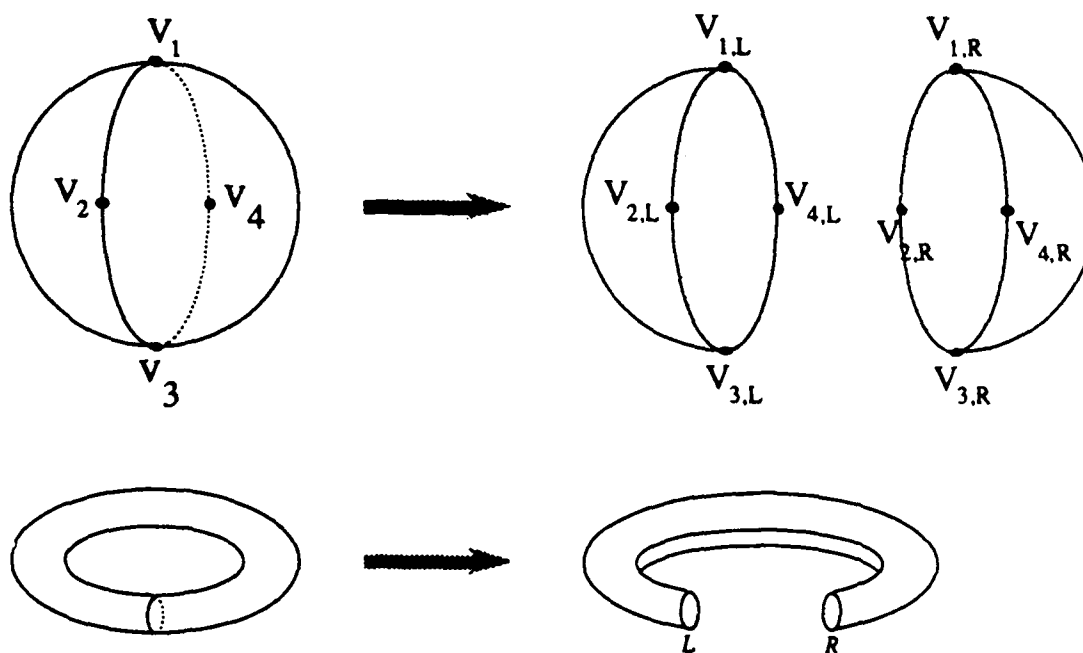


Figure 2-6: Given an embedding of a graph  $G$  on a surface of corresponding genus, we can remove a simple cycle of graph  $G$  from the surface and patch the resulting holes with disks. The result is an embedding of the graph  $G'$  obtained by splitting the given cycle of  $G$ . The new surface may or may not be connected.

Let us calculate the Euler characteristic of the new surface. The new graph  $G'$  is embedded on the new surface such that all faces are simply connected since the disks used to patch the cut are simply connected and no other faces of the original graph  $G$  are altered by the procedure. Therefore, if  $v'$ ,  $e'$ , and  $f'$  are the number of vertices, edges, and faces in the cycle-split graph  $G'$ , then the Euler characteristic of the new surface is equal to  $\chi' = v' - e' + f' = 2 - 2\gamma'$  where  $\gamma'$  is the genus of graph  $G'$  and the surface upon which it is embedded. We have that  $v' = v + l$  and  $e' = e + l$  and  $f' = f + 2$ . Therefore, we have that  $\chi' = \chi + 2$ .

First we will consider the case where the new surface is disconnected. In this case the Euler characteristic is simply the sum of the Euler characteristics of the two pieces (the formula  $v - e + f$  is additive). Let one piece have characteristic  $\chi_L = 2 - 2\gamma_L$  and the other piece have characteristic  $\chi_R = 2 - 2\gamma_R$ . Then we have that

$$\begin{aligned}\chi' &= \chi_R + \chi_L \\ &= 2 - 2\gamma_L + 2 - 2\gamma_R \\ &= 4 - 2(\gamma_L + \gamma_R).\end{aligned}$$

We also have that  $\chi' = \chi + 2 = 4 - 2\gamma$ . Therefore, we have that

$$\begin{aligned}4 - 2\gamma &= 4 - 2(\gamma_L + \gamma_R) \\ \gamma &= \gamma_L + \gamma_R\end{aligned}$$

which proves case (1).

Next we consider the case where the new surface is connected. Then we have that  $\chi' = 2 - 2\gamma'$  and  $\chi' = \chi + 2 = 4 - 2\gamma$ . Therefore, equating the right-hand sides, we have that  $2 - 2\gamma' = 4 - 2\gamma$  or  $\gamma' = \gamma - 1$  which proves case (2). ■

Now we proceed to the proof of the missing-edge lemma. First, let us remind the reader of some terminology. Bad vertices are defined as they were in section 2.5: no adjacent edge can be contracted without potentially creating a vertex with degree exceeding the bound. Bad degree-three vertices must be adjacent to three degree- $d_{\max}$  independent vertices (no edges between any pair). Thus degree-three vertices force a stronger sparsity than Euler's formula alone since these three *missing edges* can be added to the graph without increasing its genus. The missing-edge lemma quantifies this increased sparsity.

For convenience we restate the lemma as it appeared in section 2.5, assuming this time that the graph is connected. The proof goes through for disconnected graphs where genus is defined in the usual way.

**Lemma 1 (Missing-Edge Lemma)** *A graph  $G = (V, E)$  of genus  $\gamma$  with  $b_3 \geq 0$  bad degree-3 vertices has at most  $3|V| - b_3 + 10\gamma$  edges.*

*Proof.* The proof centers on showing that the number of missing edges is at least  $b_3 - 4\gamma + 1$  for  $b_3 > 0$ , which, together with the constraint that  $|E| < 3|V| + 6\gamma$  from Euler's formula, suffices to prove the lemma. The goal of the proof is to show that sharing of missing edges is limited. For the remainder of this proof, when we say "bad vertex," we assume the vertex has degree 3.

We begin our induction by showing that the lemma holds for the case  $\gamma = 0$  for any number  $b_3$  of bad vertices. That is, we show that a planar graph  $G = (V, E)$  with  $b_3 \geq 0$  bad degree-3 vertices has at most  $3|V| - b_3$  edges.

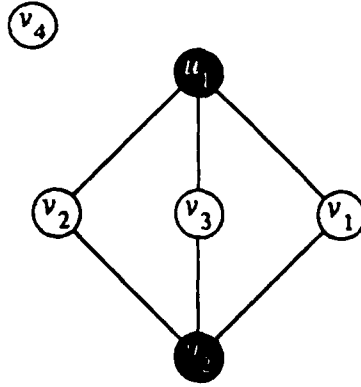


Figure 2-7: A cycle  $C = \langle u_1, v_1, u_2, v_2 \rangle$  that separates  $G$  into two subgraphs  $G_{\text{in}}$  and  $G_{\text{out}}$ , each with at most  $b_3 - 1$  bad degree-3 vertices. Nodes  $u_1$  and  $u_2$  are of maximum degree.

The proof of the planar case is by induction on  $b_3$ . If  $b_3 = 0$ , Euler's formula alone is sufficient. The lemma holds trivially for the cases  $b_3 = 1$  and  $b_3 = 2$  since any graph with at least one bad degree-3 vertex has at least three distinct missing edges. Now consider the case  $b_3 = 3$ . If there are only three missing edges in the graph, then each of the three bad vertices is associated with the same three missing edges. Hence all three bad vertices are adjacent to the same three degree- $d_{\text{max}}$  vertices, and hence the graph contains an instance of  $K_{3,3}$ , which violates the assumption that the graph is planar. Consequently, the graph has at least  $4 = b_3 + 1$  missing edges.

We now consider the general planar case  $b_3 \geq 4$ . Assume inductively that any graph with  $k < b_3$  bad vertices has at least  $k + 1$  missing edges, but that there exists a planar graph  $G = (V, E)$  with  $b_3$  bad vertices and  $m \leq b_3$  missing edges. Since each bad vertex is associated with exactly three missing edges, it follows that there exists a missing edge  $(u_1, u_2)$  associated with at least three bad vertices  $v_1, v_2$ , and  $v_3$ . There must be at least one additional bad vertex  $v_4$ , since we have  $b_3 \geq 4$ . For some embedding of  $G$  on the sphere, the Jordan Curve Theorem ensures that two of the vertices, say  $v_1$  and  $v_2$ , together with  $u_1$  and  $u_2$  form a cycle  $C$  that separates  $v_3$  from  $v_4$ , as shown in Figure 2-7.

We are now set up to apply the induction hypothesis. Let  $G_{\text{in}}$  be the subgraph of  $G$  induced by cycle  $C$  and the vertices on one side of  $C$ , and let  $G_{\text{out}}$  be the subgraph induced by cycle  $C$  and the vertices on the other side of  $C$ . Add to  $u_1$  and  $u_2$  in each of  $G_{\text{in}}$  and  $G_{\text{out}}$  enough degree-1 neighbors to maintain their degrees as  $d_{\text{max}}$ . Let  $b_{\text{in}}$  be the number of bad (degree-3) vertices in  $G_{\text{in}}$ , and let  $b_{\text{out}}$  be the number of bad vertices in  $G_{\text{out}}$ . Since no new bad degree-3 vertices are introduced into  $G_{\text{in}}$  and  $G_{\text{out}}$ , we have  $b_{\text{in}} < b_3$  and  $b_{\text{out}} < b_3$ . By the induction hypothesis, therefore, graph  $G_{\text{in}}$  has at least  $b_{\text{in}} + 1$  missing edges, and graph  $G_{\text{out}}$  has at least  $b_{\text{out}} + 1$  missing edges.

We next account for interactions between graphs  $G_{\text{in}}$  and  $G_{\text{out}}$  to obtain the lower bound of  $b_3 + 1$  on the number of missing edges in the original graph  $G$ . First, observe that  $b_{\text{in}} + b_{\text{out}} = b_3$ , since when  $v_1$  is a bad vertex in one of the graphs, it is a degree-2 vertex in the other, and similarly for  $v_2$ . Moreover, each of  $v_1$  and  $v_2$  are bad in at least one of  $G_{\text{in}}$  and  $G_{\text{out}}$  because we dummed up each of  $u_1$  and  $u_2$  to have degree  $d_{\text{max}}$ . The number of missing edges in  $G$  is at least  $(b_{\text{in}} + 1) + (b_{\text{out}} + 1)$  minus the number of missing edges shared by  $G_{\text{in}}$  and  $G_{\text{out}}$ . The

number of such shared missing edges is in fact 1, namely, the edge  $(u_1, u_2)$ , since  $u_1$  and  $u_2$  are the only degree- $d_{\max}$  vertices in both  $G_{\text{in}}$  and  $G_{\text{out}}$ . Consequently, the number of missing edges in  $G$  is at least  $(b_{\text{in}} + 1) + (b_{\text{out}} + 1) - 1 = b_3 + 1$ , which completes proof of the planar version of the lemma.

We have shown that the missing edge lemma holds for the case  $\gamma = 0$  for any number  $b_3$  of bad vertices. To complete the base cases, we see that for general graphs the lemma holds for  $b_3 = 0$  trivially using only Euler's formula. It also holds trivially for  $1 \leq b_3 < 4\gamma + 3$  for all  $\gamma$  since then the lemma only requires that the number of missing edges is at least 3 which is true for any graph with at least one bad vertex.

We now consider the general case  $b_3 \geq 4\gamma + 3$  and  $\gamma \geq 1$ . We assume inductively that any genus- $\gamma$  graph with  $k < b_3$  bad vertices has at least  $k - 4\gamma + 1$  missing edges and any graph with genus  $j < \gamma$  and any number  $b_3$  bad degree-three nodes has at least  $b_3 - 4j + 1$  missing edges. Assume, however that there exists a graph  $G = (V, E)$  of genus  $\gamma$  with  $b_3 \geq 4\gamma + 4$  bad vertices and  $m \leq b_3 - 4\gamma$  missing edges. Since each bad vertex is associated with exactly three missing edges, it follows that there exists a missing edge  $(u_1, u_2)$  associated with at least three bad vertices  $v_1, v_2$ , and  $v_3$ . There must be at least one additional bad vertex  $v_4$  since we have  $b_3 \geq 7$ . The situation is illustrated in figure 2-7.

To allow application of the induction hypothesis, we split graph  $G$  along one of the three simple cycles shown in figure 2-7 to obtain a new graph  $G'$  of genus  $\gamma'$ . As we did in proving the planar version of this lemma, we add degree-one neighbors to the high-degree nodes  $u_1$  and  $u_2$  so that each of the four nodes resulting from the split of  $u_1$  and  $u_2$  has maximum degree. Each bad degree-three node on the cycle is split into two nodes, but exactly one of these two nodes is a bad degree-three node in the new graph  $G'$ . The other node split from that vertex is of degree 2. Therefore graph  $G'$ , whether connected or not, has exactly  $b_3$  bad nodes.

Suppose we can split graph  $G$  along one of the cycles such that the resulting graph  $G'$  is still connected. Then by lemma 6(2) we have that  $\gamma' = \gamma - 1$ . As argued above, graph  $G'$  has the same number  $b_3$  of bad degree-three vertices as the original graph did. Because graph  $G'$  has lower genus, we can apply the induction hypothesis. Therefore, graph  $G'$  has at least  $b_3 - 4(\gamma - 1) + 1 = b_3 - 4\gamma + 5$  missing edges. We overcounted exactly one missing edge, namely  $(u_1, u_2)$ , since these are the only two degree- $d_{\max}$  vertices duplicated. Therefore we have at least  $b_3 - 4\gamma + 4 > b_3 - 4\gamma + 1$  missing edges.

Now suppose that splitting along any of the three simple cycles separates  $G'$  into two graphs: graph  $G_L$  with genus  $\gamma_L$  and  $b_L$  bad degree-three vertices, and graph  $G_R$  with genus  $\gamma_R$  and  $b_R$  bad degree-three vertices. Suppose that splitting along one of the cycles yields graphs such that  $\gamma_L < \gamma$  and  $\gamma_R < \gamma$ . Applying the induction hypothesis to each side we have that the number of missing edges in graph  $G_R$  is at least  $b_R - 4\gamma_R + 1$  and the number of missing edges in graph  $G_L$  is at least  $b_L - 4\gamma_L + 1$ . Adding the missing edges together and subtracting one for the overcount of edge  $(u_1, u_2)$ , we have that the original graph  $G$  has at least  $(b_L + b_R) - 4(\gamma_L + \gamma_R) + 2 - 1$  missing edges. From lemma 6(1) we have that  $\gamma_L + \gamma_R = \gamma$  and we argued earlier that  $b_L + b_R = b_3$ . Substituting these back into the expression for the number of missing edges in graph  $G$ , we have that graph  $G$  has at least  $b_3 - 4\gamma + 1$  missing edges.

The final case we must consider is that each of the three cycles cuts off a planar patch. In this case we have a situation analogous to the planar case argued earlier. As illustrated in figure 2-7, one of the cycles cuts the graph into two graphs  $G_L$  and  $G_R$  such that  $b_L < b_3$  and  $b_R < b_3$ . Without loss of generality, let us assume that graph  $G_R$  is planar. Then by lemma 6(1),

graph  $G_L$  must have genus  $\gamma$ . Applying the induction hypothesis to the two graphs (since each has less than  $b_3$  bad vertices), we have that  $G_R$  has at least  $b_R + 1$  missing edges and  $G_L$  has at least  $b_L - 4\gamma + 1$  missing edges. Again we have that  $b_R + b_L = b_3$ . Adding the missing edges and subtracting one for the overcount of edge  $(u_1, u_2)$ , we have that graph  $G$  has at least  $b_L + b_R - 4\gamma + 2 - 1 = b_3 - 4\gamma + 1$  missing edges which completes the proof. ■

## 2.7 Applications

The most direct application of the parallel contraction algorithm is to the vision problem of determining the connected regions of an image represented as a two-dimensional array of pixels. We can view the image as a planar graph  $G = (V, E)$ , where the vertex set  $V$  is the set of pixels, and  $(u, v) \in E$  if  $u$  and  $v$  are adjacent pixels with the same color. The *region labeling* problem is to assign each pixel an integer such that two pixels have the same label if and only if they are path connected in  $G$ .

Region labeling can be solved by a connected-components algorithm. Shiloach and Vishkin [111] give an  $O(\lg n)$ -time,  $n$ -processor parallel algorithm for connected components using the concurrent-read, concurrent-write (CRCW) PRAM model. Hagerup [54] gives an  $O(\lg n)$ -time,  $n/\lg n$ -processor CRCW algorithm for graphs of bounded genus<sup>1</sup>. The best algorithm to date in the more easily implemented exclusive-read, exclusive-write (EREW) model is due to Hirschberg, Chandra, and Sarwate [60], who give an  $O(\lg^2 n)$  time algorithm for connected components using the adjacency-matrix representation of a graph. Leiserson and Maggs [78] give an  $O(\lg^2 n)$  step,  $n$ -processor, randomized connected-components algorithm for a DRAM (distributed random-access machine), an EREW-like model that includes the cost of communication. Blelloch [14] gives an  $O(\lg n)$  randomized algorithm for a model that includes parallel prefix as a primitive operation. Lim [82] gives a region-labeling algorithm that runs in  $O(\lg^2 n)$  time on an EREW PRAM which uses the planar and geometric properties of a mesh. Gazit and Reif have recently developed a deterministic  $O((n + m)/\lg n)$ -processor EREW algorithm that runs in time  $O(\lg n + \lg^2 g)$  where  $g$  is the genus of the graph (what we call  $\gamma_G$ , which can be much larger than  $\gamma$  in disconnected graphs). They require an embedding of the graph. Other algorithms for connected components are given by [25,45,74,91,122,127].

Our algorithm for labeling planar graphs is asymptotically the fastest algorithm to date in the EREW PRAM model. The algorithm uses Algorithm BOUNDED-CONTRACT to simultaneously contract each component of the graph to a single node, and then it simply reverses the contraction process and assigns the same label to all nodes in each connected component. The algorithm uses  $O(n/\lg n)$  processors, and with high probability (i.e., probability at least  $1 - O(1/n^k)$  for any constant  $k$ ), the algorithm runs in  $O(\lg n)$  time on an EREW PRAM or a DRAM. Thus, our algorithm achieves the best possible asymptotic bound in the most restrictive parallel models. The deterministic version of the algorithm runs in  $O(\lg n \lg^* n)$  time using an optimal number of processors  $(n/\lg n \lg^* n)$ . Like Lim's  $O(\lg^2 n)$ -time algorithm, our algorithm for region labeling takes advantage of the planar nature of the image graph, but unlike his algorithm, it does not depend on the geometric nature of the image.<sup>2</sup> Thus, our algorithm

<sup>1</sup>Hagerup has recently independently developed an  $O(\lg n \lg^* n)$ -time  $n/\lg n \lg^* n$ -processor deterministic EREW algorithm for graphs of bounded genus [55]

<sup>2</sup>Subsequent to the development of the planar version of our algorithm, Lim, Agrawal, and Nekudova [83]

works on irregular planar structures, including meshes with local refinements and meshes with (static) faulty elements.

The contraction algorithm can also be used for a variety of other applications including algorithms for biconnected components of planar graphs and for spanning trees of planar graphs. The running times for these algorithms is asymptotically the same as for the contraction algorithm, and for these problems the running times are asymptotically the best to date in the EREW model. All the algorithms, including the connected components algorithm, can be generalized to graphs of higher genus and higher degree using algorithm BOUNDED-CONTRACT. The asymptotic running times for these problems match the running times of the contraction algorithm (i. e.  $O(\lg n + \lg^2 \gamma)$ ) with high probability where  $\gamma$  is the maximum genus of any connected component). For classes of graphs with genus  $o(n^\epsilon)$  for constant  $\epsilon > 0$ , this is the best bound to date. For graphs of genus  $\Omega(n^\epsilon)$  this matches the best previous bound of  $O(\lg^2 n)$ .

## 2.8 Conclusion

In this section we comment upon the practical behavior of algorithm BOUNDED-CONTRACT, discuss its application to graphs of unbounded degree, and present some open problems.

In section 2.5 we commented that the behavior of algorithm BOUNDED-CONTRACT in practice is likely to be much better than that guaranteed by our worst-case analysis. This is particularly true for the bounded-genus case where we explicitly check for termination. For example, we are not likely to require the worst-case number of iterations to reduce each component of a planar graph to a single node. The general algorithm, however, always performs the worst-case number of iterations for phase 1 because we do not have a means of detecting when phase 1 is completed unless we in fact complete the entire contraction. Phase 2 only performs as many contraction steps as necessary. Because of this control structure, it is probably good in practice to choose the constant  $j$  in line 1 of algorithm CONTRACT based upon lemma 2 with a fraction of  $1/c$  good nodes for constant  $c = 6 + \epsilon$  and  $\epsilon > 0$ .

If we are primarily interested in the practical behavior of an application of contraction such as connected components, we can combine phase 1 of algorithm BOUNDED-CONTRACT with any  $O(\lg^2 n)$ -time EREW algorithm for connected components. Although we will not be performing contraction to completion, we will compute the connected components in the same asymptotic time and we may achieve better practical performance by using a different approach to phase 2.

We can apply algorithm BOUNDED-CONTRACT to any graph by using the maximum degree of the graph  $d_{\max}$  in place of 40 in line 3 of the algorithm. The algorithm will contract the graph in time  $O(\lg n \lg d_{\max} + \lg^2 \gamma)$  where  $\gamma$  is the component genus. Lemmas 1 and 2 apply to any graph since eligible edges (and good vertices) are defined relative to the maximum degree  $d_{\max}$ . Theorem 5 also applies to any graph by changing the degree 40 to degree  $d_{\max}$ . Then the analysis of the running time proceeds the same as the analysis in section 2.5 except that the time needed to execute a parallel paired contraction step is now  $\lg d_{\max}$ , which is still constant time if  $d_{\max}$  is bounded. The corresponding bound for the deterministic algorithm is  $O(\lg n (\lg d_{\max} + \lg^2 n) + \lg^2 \gamma)$  which is asymptotically the same as the randomized algorithm

---

obtained an  $O(\lg n)$ -time deterministic algorithm for region labeling. Like Lim's earlier work, and unlike ours, their algorithm exploits the geometric properties of specific planar grids.

unless  $d_{\max} < \lg^* n$ ). Thus, continuing the discussion at the end of section 2.7, our algorithm is asymptotically the best to date for any graph with genus  $o(n^\epsilon)$  and degree  $o(n^\epsilon)$  for any constant  $\epsilon > 0$ . The restriction on degree is removed if we have an embedding.

To detect the termination of phase 1 we require an  $O(\lg n)$  time  $n$ -processor algorithm which can determine the genus of an  $n$ -node graph to within a constant without knowing the value of  $n$ . The problem appears difficult since determining the exact genus  $\gamma$  of an  $n$ -node graph is NP-complete [117] and the best sequential algorithm for this problem runs in  $O(n^{O(\gamma)})$  time [40]. There seems to be no work on parallel algorithms which approximate the genus of a graph.

## Chapter 3

# Vector-Matrix Primitives

### 3.1 Introduction

When implementing a parallel algorithm, it is convenient to have a high-level parallel language which provides the convenience one has come to expect from well-established serial languages. One wishes to concentrate on the details of the algorithm, allowing the language to abstract away details of the machine including the number of processors, the interconnection network, and the embedding of data elements into processors. However, since parallel machines are currently very expensive and used for huge, computationally intensive applications, users often will not give up performance for ease of programming and portability. Techniques for mapping high-level descriptions of algorithms onto efficient code for various parallel machines have therefore become very important and will probably consume a large portion of computer science research in the next decade.

This chapter provides such a technique by showing how very high-level descriptions of a broad class of dense matrix algorithms can be mapped onto a real machine, the Connection Machine, with no performance loss over hand coded versions. It presents four high-level APL-like primitives and illustrates code for a vector-matrix multiply, a Gaussian-elimination routine and a simplex algorithm for linear programming based on these primitives. The algorithms are straight forward implementations of the classic algorithms. The code is high-level, it works for any sized matrices<sup>1</sup>, and contains no information on how data is mapped onto processors nor on how the data should be communicated. Therefore it is concise: none of our routines contain more than 20 lines of code. The chapter then discusses our implementation of the primitives on the Connection Machine and gives actual timings for both the primitives and two of the algorithms. The simplex and vector-matrix multiply execute faster than any other code for these applications for the Connection Machine. With 256 matrix elements per processor, an iteration of simplex runs at 525 Mflops and Matrix-Vector multiply runs at over 1 Gflop on a 64K CM-2 (single precision). Versions of the primitives we implemented are now included in PARIS, the parallel instruction set of the Connection Machine.

The four primitives we consider *extract* a vector from a row or column of a matrix, *insert* a vector into a row or column, *distribute* a vector across the rows or columns, and *reduce* the rows or columns into a vector using a binary associative operator such as +, maximum, or minimum.

---

<sup>1</sup>Our current implementation is restricted to sizes which are powers of 2.

Figure 3-1 illustrates the semantics of these four primitives. It is convenient to also use an operation which *spreads* a row or column across its matrix (*extract* followed by *distribute*). We also assume the existence of various other simple primitives—these primitives are summarized in Figure 3-2.

Although the *extract* and *insert* primitives might seem trivial, their implementation is actually as complex as the implementation of the *distribute* and *reduce* primitives. They involve rearranging the data among the processors by communicating along trees embedded within subcubes occupied by the rows or columns of a matrix. This data transfer guarantees optimal load balancing of any row or column vector extracted from a matrix. Thus after extraction each processor holds (within one) the same number of vector elements ( $= \lceil m/p \rceil$  where  $m$  is the number of matrix elements and  $p$  is the number of processors). Load balancing the vectors improved the performance of the simplex algorithm from under 100MFlops to over 500MFlops.

There has been considerable research on implementing dense matrix algorithms on hypercube-based machines [89,47,65,42,22,23]. This chapter concentrates on how to get similar results without having to code for the particular machine. The final execution of the linear systems solver is similar to the algorithm suggested for a hypercube by Johnsson [65]. Our implementations of the primitives are simple, clean subcases of the  $n$  edge-disjoint spanning binomial trees (NSBT) algorithms due to Johnsson and Ho [66]. Our *extract* implementation is a version of what they call one-to-all personalized communication within subcubes and our *distribute* implementation is an example of all-to-all broadcasting. Other related algorithms for hypercubes are discussed by Fox and Furmaski [43], Stout and Wager [113], and Deshpande and Jenevin [35].

To motivate some of the decisions we made in the selection and implementation of primitives, let us examine, as an example, the solution of linear programs using the Dantzig simplex method. The standard form of a linear programming problem is as follows:

$$\text{minimize } c^T x \quad \text{such that} \quad \begin{cases} Ax = b \\ x \geq 0 \end{cases}$$

where  $c$  is an  $m_2$ -dimensional integer *objective function* vector,  $A$  is an  $m_1 \times m_2$  integer *constraint matrix*,  $b$  is an  $m_1$ -vector of integers, and  $x$  is a real  $m_2$ -vector of unknowns. All information needed to perform a step of the computation is kept in a *tableau* which is initially the constraint matrix  $A$  augmented by the column vector  $b$  and the row vector  $c$ . A single step of the computation is then one step of Gaussian elimination on the entire tableau where the pivot column has the most negative value of vector  $c$  and the pivot row has the smallest positive value of  $b/a$  within the pivot column. Section 3.2 explains some of the ideas behind the method—for now, an understanding of the functionality suffices.

If we have as many processors as tableau elements, a straight-forward implementation of one simplex step can be written as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$B = [ b_1 \quad b_2 \quad \dots \quad b_n ]$$

$$\text{extract-row}(A, 2) = [ a_{21} \quad a_{22} \quad \dots \quad a_{2n} ]$$

$$\text{deposit-row}(A, B, 1) = \begin{bmatrix} b_1 & b_2 & \dots & b_n \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$\text{distribute-row}(B) = \left[ \begin{bmatrix} b_1 & b_2 & \dots & b_m \\ b_1 & b_2 & & b_m \\ \vdots & & \ddots & \vdots \\ b_1 & b_2 & \dots & b_m \end{bmatrix} \right] n$$

$$\begin{aligned} &+-\text{reduce-to-row}(A) = \\ &[ a_{11} + a_{21} + \dots + a_{m1}, \quad a_{12} + a_{22} + \dots + a_{m2}, \quad \dots, \quad a_{1n} + a_{2n} + \dots + a_{mn} ] \end{aligned}$$

Figure 3-1: The four primitives we consider *extract* a vector from a row or column of a matrix, *insert* a vector into a row or column, *distribute* a vector across the rows or columns, and *reduce* the rows or columns into a vector using a binary associative operator such as +, maximum, or minimum. The number of columns  $n$  across which a row is distribute'd is determined by context, usually the dimensions of the matrix variable which is set to the result of the distribute:  $C = \text{distribute-row}(B)$ .

Scalar Instructions:  $+$ ,  $-$ ,  $\times$ ,  $\dots$   
 Global Instructions: Broadcast, g-min, g-max,  $\dots$   
 Elementwise Vector and Matrix Instructions:  $p+$ ,  $p-$ ,  $p*$ ,  $p\div$ ,  $\dots$   
 Vector-Matrix Instructions: insert, extract, distribute, reduce

Figure 3-2: The instructions we use in the algorithms described in this chapter. *Global* instructions compute a single value from all elements of a vector or matrix or broadcast a single value across a vector or matrix. *Elementwise* instructions perform an operation elementwise over corresponding elements of equal sized matrices or vectors.

#### Algorithm Simplex

```

;tableau  $T ((m_1 + 1) \times (m_2 + 1))$ 
;initially matrix  $A$  augmented by
;column vector  $B$  and row vector  $C$ 

repeat forever:
1   selecting processors that initially held vector  $C$ 
2    $pivotcolnum =$  column # of processor holding g-min( $T$ )
      (if g-min( $T$ )  $\geq 0$ , exit Simplex successfully)
3   selecting processors that initially held vector  $B$ 
4   send value of  $T$  to pivot column (store locally as  $B$ )
5   selecting processors in Pivot column with  $T > 0$ 
      (if none, exit Simplex unsuccessfully)
6    $Ratio = p\div(B, T)$ 
7    $pivotrownum =$  row # of processor holding g-min( $Ratio$ )
8    $pivotelement = A[pivotcolnum][pivotrownum]$ 
9   selecting processors in the pivot row
10   $T = p\div(T, Broadcast(pivotelement))$ 
11  send  $T$  value of pivot row to all rows (store in  $Pivotrow$ )
12  send  $T$  value of pivot column to all columns (store in  $Pivotcol$ )
13   $T = p-(T, p*(Pivotrow, Pivotcol))$ 

```

In general, however, we would like to run problems in which the tableau is larger than the number of processors. Figure 3-3 shows how we might map an  $m_1 \times m_2$  matrix onto a machine where the processors can be logically configured as a grid. Each processor holds a  $k_1 \times k_2$  submatrix which is as square as possible (an aspect ratio of at most 2). The runtime environment can keep track of how a matrix is mapped onto the processors, and can automatically loop over all the elements in each processor when operating on the matrix. This bookkeeping is best executed at run-time rather than compile-time because the executable code should be independent of the number of processors in a machine.

This embedding is *load balanced* with respect to the matrix since each processor holds an equal number of matrix elements. Any given row or column, however, when viewed as a vector is

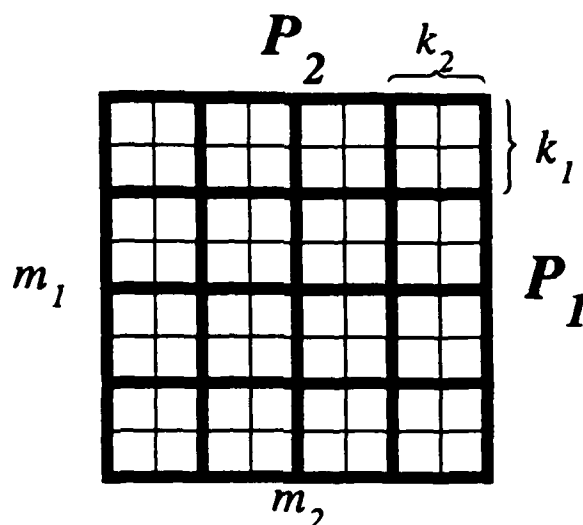


Figure 3-3: Each processor in an  $P$ -processor hypercube viewed as a  $p_1 \times p_2$  grid of processors holds a  $k_1 \times k_2$  submatrix of the  $m_1 \times m_2$  matrix ( $P = p_1 p_2$ ,  $m_1 = p_1 k_1$  and  $m_2 = p_2 k_2$ ). In this example,  $k_1 = k_2 = 2$ ,  $p_1 = p_2 = 4$ , and  $m_1 = m_2 = 8$ . Heavy lines represent processor boundaries and thin lines represent matrix element boundaries.

not well-balanced at all as shown in Figure 3-4. In particular, when performing the divisions in lines 6 and 10 and the minimums in lines 2 and 7 in the simplex algorithm above, a small subset of the processors must perform many computations while the rest of the processors are idle. When computing on a single column, as in line 6, it may be more efficient to use the underlying interconnection network to spread the work from the one active processor in each row to the idle processors in its row. For example, in computing on the column selected in Figure 3-4, the two active processors send one operation to each of the three idle processors in their rows so that each processor performs one operation. Furthermore, it may be beneficial to always leave the  $B$  and  $C$  vectors stored in this load-balanced fashion and to update them in place. In our implementation on the Connection Machine System, a carefully coded implementation similar to the code above ran at 55 Mflops. After adding the load balancing, the loop ran at 525 Mflops. Furthermore, the code as shown in section 2.7 no longer contains the awkward selection statements since logical vector and matrix functionalities have been separated.

In seeking to generalize from this experience, we found that the four classes of primitives provide expressive power and potentially automatic efficiency improvement. In the simplex example, our inefficiencies came from treating vectors as matrices. Distinguishing between vector and matrix computations is not only efficient, but also fits well into the way programmers think about computing on these objects.

Section 3.2 gives examples of the use of the primitives in several applications including the simplex method for linear programming. The four classes of primitives provide cues that computation is shifting from a full matrix to a vector related to that matrix (eg. a row or column), indicating that it may be a proper time for load balancing. Bookkeeping related to the looping over matrix or vector elements within a processor is best executed at runtime rather than compile time because the binary code should be independent of the number of processors

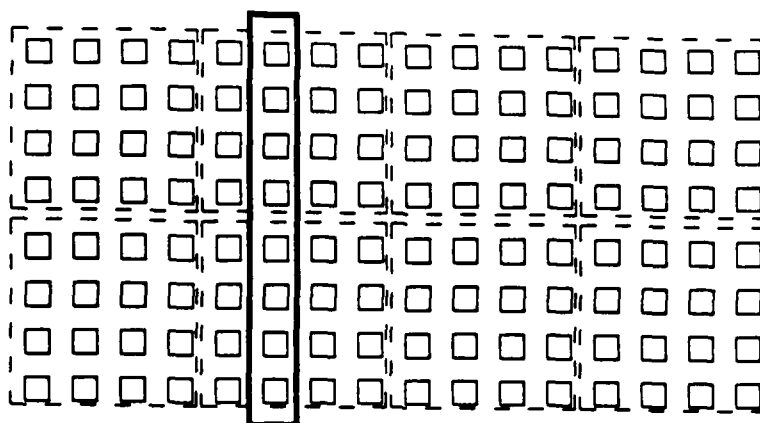


Figure 3-4: All the elements of a single column are located in a small subset of the processors. In the example they are located in 2 of the 8 processors. When extracting a column, the elements are divided among the processors so that each processor gets one element of the column.

in a machine. A compiler, however, can frequently decide whether it is better to load balance during an extract or reduce operation or to leave the elements in place.

Section 3.3 discusses the implementations of the primitives for *hypercube* multiprocessors, described in section 1.3. The algorithms we present in this chapter work on any hypercube multiprocessor, but they are simple enough to run on a *data parallel architecture* in which each processor executes the same instruction broadcast from a *front end* computer (also known as a single instruction multiple data architecture).

Remarkably, although our implementations are very simple, we prove in Section 3.3 that the *reduce* and *distribute* primitives are optimal for a hypercube in two senses: parallel time, and processor-time product (work), provided the matrix is sufficiently large relative to the number of processors and each processor is capable of sending only a constant number of messages in unit time (this model is called a *weak hypercube*).

Section 3.4 gives timings for an implementation of our primitives on the Connection Machine System [57]. The current implementations are restricted in two ways. First of all, the number of rows and columns in a matrix must be a power of 2. Secondly, row vectors and column vectors are not interchangeable. Thus a row extracted from one matrix can only be deposited into or distributed across the rows of another matrix even if the column size matches the row vector length. In section 3.3.6 we discuss techniques for avoiding the first restriction. The Connection Machine has a router for arbitrary interprocessor communication and its processors have the ability to indirectly address their memory; our algorithms, however, do not require the power of these features because of the regularity of dense matrices. Efficient sparse matrix techniques for the Connection Machine require both the router and indirect addressing (see for example chapter 5 and [16]).

### 3.2 Example Applications

In this section, we present CM implementations of matrix-vector multiplication, LU decomposition with partial pivoting, solution of a linear system from an  $LU$  decomposition and a vector  $b$ , and a simplex method for linear programming. These have been implemented using the primitives we described and are competitive with all other CM implementations to date.

#### 3.2.1 Matrix-Vector Multiply

The first example we present is a matrix-vector multiply. Using a *distribute*, we spread the input vector over all rows of the matrix, multiply the two element-wise, and reduce all rows of the result by summing across the rows, returning a column vector containing the matrix-vector product. Thus,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

becomes

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_2 & \dots & b_n \\ b_1 & b_2 & & b_n \\ \vdots & & & \vdots \\ b_1 & b_2 & \dots & b_n \end{bmatrix}$$

where the symbol  $\cdot$  represents pairwise multiplication. The result of these parallel elementwise multiplications are taken along the rows to yield the final result.

One complication is that our algorithm accepts a row vector as input but returns a column vector. This condition causes less of a problem if we want to immediately use the vector in another matrix-vector multiplication, as we would if we were investigating a Markov process. At some cost in storage, we can store both the transition matrix and its transpose, then alternate which one we use. Alternatively, we can pay  $O(f(k_1, k_2) + \lg n)$  time to transpose either the vector or the matrix, where  $f(k_1, k_2)$  is a function which depends on the routing mechanism used for the transpose and the type of object being transposed.

#### *Algorithm Matrix-Vector Multiply (MVM)*

```

;matrix A ( $m_1 \times m_2$ )
;vector B
;temporary matrix TEMP ( $m_1 \times m_2$ )

1   Temp = distribute-row(B)
2   Temp = p-*(A, Temp)
3   return(+reduce-to-column(Temp))

```

### 3.2.2 Linear System Solution by LU Decomposition

The next example demonstrates solution of a linear system by LU decomposition with partial pivoting and back solving. We start with the basic formula  $Ax = b$ , where  $A$  is a given  $m \times m$  matrix,  $b$  is a given  $m$ -vector, and  $x$  is an  $m$ -vector of unknowns. We compute two matrices  $L$  and  $U$  such that  $A = LU$ , where  $L$  is lower triangular and  $U$  is upper triangular (hence the name LU decomposition). Now we have  $LUx = b$ , which we rewrite as  $Ly = b$ . Solving for  $y$  is very simple since  $L$  is triangular [101]. This step is called *forward solution*. We then solve  $Ux = y$  which is equally easy, as  $U$  is also triangular. This step is called *back solution*. See a numerical algorithms text such as [101] for more complete descriptions of forward solution, backward solution, and partial pivoting. Since for a given system  $A$  we may wish to solve  $x$  for multiple  $b$  vectors, we split the linear solver into two algorithms, namely LU decomposition and solution. As the  $L$  and  $U$  matrices do not share any non-zero elements, we can conveniently store both matrices in a single matrix, known as the *LU matrix*.

When performing LU decomposition by Gaussian elimination, we select a row and column at each step to eliminate. The process of selecting a row is called *pivoting*. The partial pivoting technique processes the columns in left-to-right order, and selects the element of the column with the greatest absolute value. This pivot choice improves numerical stability. Rather than physically exchanging the two rows, which would yield a true lower and upper triangular matrix, we record in a separate vector which row was eliminated in which iteration. This defines a "logical" diagonal to the left and right of which reside the  $L$  and  $U$  matrices respectively. For example, the following permutation vector corresponds to the following permuted LU decomposition, where elements labeled  $L$  are part of the  $L$  matrix, those labeled  $U$  are part of the  $U$  matrix, and those labeled  $D$  are on the diagonal:

$$P = \begin{bmatrix} 2 \\ 4 \\ 1 \\ 3 \end{bmatrix}, LU = \begin{bmatrix} L_{12} & D_{22} & U_{32} & U_{42} \\ L_{14} & L_{24} & L_{34} & D_{44} \\ D_{11} & U_{21} & U_{31} & U_{41} \\ L_{13} & L_{23} & D_{33} & U_{43} \end{bmatrix}$$

The unpermuted LU decomposition looks like this:

$$LU = \begin{bmatrix} D_{11} & U_{21} & U_{31} & U_{41} \\ L_{12} & D_{22} & U_{32} & U_{42} \\ L_{13} & L_{23} & D_{33} & U_{43} \\ L_{14} & L_{24} & L_{34} & D_{44} \end{bmatrix}$$

The parallel algorithm for LU decomposition is very simple. For an  $m \times m$  matrix, we perform  $m$  steps of Gaussian elimination. Moving left to right through the matrix, we extract the columns in sequence. We find the element with the greatest absolute value, and extract its row. We then divide the pivot column by the pivot element itself, and distribute the pivot row and column across the matrix. We replace the part of the pivot column logically below the pivot element, where it will serve as a column of  $L$ . Finally, we perform Gaussian elimination on the square logically below and to the right of the pivot row and column. Note that the selections in step 2 and 12 are nested; the processors selected in step 12 are a subset of those selected in step 2.

*Algorithm LU Decomposition (LUD)*

```

;matrix A ( $m \times m$ )
;LU decomposition with partial pivoting
;Returns LU in A, permutation vector in P

1  For i from 1 to m Do
2      selecting rows and columns of A that have not been pivoted on
3          Col = extract-column(A, i)
4          pivotrow = row # of g-max(p-abs(A))
5          Row = extract-row(A, pivotrow)
6          P[pivotrow] = i
7          pivotelement = A[pivotrow, i]
8          Col = p-÷(Col, Broadcast(pivotelement))
9          A = deposit-column(A, Col, i)
10         Colmatrix = distribute-column(Col)
11         Rowmatrix = distribute-row(Row)
12         selecting positions in A that are not in the pivot row or column
13         A = p--(A, p-*(Rowmatrix, Colmatrix))

```

The forward and back solution phase is also straightforward. Given a vector  $b$ , we first solve  $Ly = b$ , divide  $y$  and  $U$  through by the logical diagonal such that the system has a unit diagonal, and then solve  $Ux = y$ . A solution step consists of extracting a column, multiplying it by the diagonal element, and subtracting from  $b$ . Note that to find the pivot element we compare the permutation vector against the loop index.

*Algorithm Solve Linear System from Decomposition*

```

;matrix LU ( $m \times m$ )
;vector B
;permutation vector P
;Solve  $LUx = B$ 

1  For i from 1 to m Do
2      Column = extract-column(LU, i)
3      pivot = B[k] such that  $i = P[k]$ 
4      selecting the elements B[j] and Column[j] such that  $P[j] > i$ 
5      B = p--(B, p-*(Column, Broadcast(pivot)))
6      send logical diagonal of LU to first col of Temp
7      Diag = extract-column(Temp, 1)
8      ;At this point B contains y from  $Ly = b$ 
9      ;Divide U and B by the diagonal.
10     B = p-÷(B, Diag)
11     Temp = distribute-column(Diag)
12     Selecting positions in the logical upper triangle of LU

```

```

11      LU = p-÷(LU, Diag)
12  For i from m downto 1 Do
13      Column = extract-column(LU, i)
14      pivot = B[k] such that P[k] = i
15      selecting the elements B[j] and Column[j] such that P[j] ≤ i
16      B = p-÷(B, p-*(Column, Broadcast(pivot)))
17  unpermute B

```

### 3.2.3 Simplex method for linear programming

Our final example illustrates the Simplex method for solving linear programming problems. The standard form of a linear programming problem is as follows:

$$\text{minimize } c^T x \quad \text{such that} \quad \begin{cases} Ax = b \\ x \geq 0 \end{cases}$$

where  $c$  is an  $m_2$ -dimensional integer *objective function* vector,  $A$  is an  $m_1 \times m_2$  integer *constraint matrix*,  $b$  is an  $m_1$ -vector of integers, and  $x$  is a real  $m_2$ -vector of unknowns. Generally we have  $m_1 < m_2$ .

A vector  $x$  such that  $Ax = b$  and  $x \geq 0$  is called a *feasible solution* because it satisfies all the constraints. If a linear program has an optimal solution, we can always find one such that  $m_1$  of the entries in vector  $x$  are equal to 0 [96]. Such vectors, called *basic feasible solutions*, correspond geometrically to corners of the convex  $(m_2 - m_1)$ -dimensional polytope of all feasible solutions. The simplex method for solving linear programs due to Dantzig [33] starts at a basic feasible solution and *pivots* to a new basic feasible solution which improves the objective function. Algebraically, we increase one of the zero-valued *nonbasic* variables (the *entering variable*) until one of the non-zero *basic* variables becomes zero. In the Dantzig method of pivoting, the entering variable is the one that will decrease the objective function by the most (per unit increase in the variable).

All the information necessary to perform the pivoting is kept in a *tableau* where the objective function and all nonbasic variables are represented in terms of the basic variables. At the start, the tableau is the constraint matrix  $A$  augmented by the column vector  $b$  and the row vector  $c$ . Vectors  $b$  and  $c$  are also maintained in vector representation. We use Gaussian elimination to eliminate all columns corresponding to basic variables. Since the columns that correspond to the basic variables always form the identity matrix, we can save space by not representing these columns at the cost of some extra time per iteration. We also need an extra front-end array to indicate the mapping of variables to rows and columns. The code below and in Section 3.1 does not include this optimization although the code used to gather the timings does.

Since all the nonbasic variables are zero at the basic feasible solution represented by the tableau, the  $b$  vector represents values of the basic variables and objective function at the basic feasible solution and the objective function vector  $c$  represents the unit change in the objective function per unit increase in each nonbasic variable. To form the tableau for which one basic variable is replaced by a nonbasic variable then involves one step of Gaussian elimination.

The tableau representation is used primarily for linear programs for which the constraint matrix  $A$  is *dense*. In practice many linear programs from real applications are sparse. Implementations on sequential computers use special techniques to avoid computing on the whole

matrix when only a few elements are non-zero. When the matrix is dense, however, the tableau method (or the revised method which is more numerically stable) can be practical.

The implementation of Simplex with Dantzig's rule is fairly straightforward. We first find the index of the most negative coefficient of the objective function; pivoting on this variable will give us the most rapid improvement in the solution per unit increase in the entering nonbasic variable. If there are no negative coefficients, then we cannot make any improvement, and thus have finished successfully. We then extract the indexed column, and select the positions corresponding to real constraints, i. e. only positive coefficients correspond to basic variables that decrease as the entering variable increases. If there are no positive coefficients in the column, then the system is unbounded; we can increase the value of variable improving the objective function without limit and never violate a constraint. To find the limiting constraint, we divide the  $b$  vector by the positive elements of the pivot column elementwise and find the index of the smallest ratio. The two indices define the pivot element. We then perform a Gaussian elimination step.

#### *Algorithm Simplex*

```

;tableau A ((m1 + 1) × (m2 + 1))
;constraint vector B
;objective function vector C

repeat forever:
1   pivotcolnum = col # of element holding g-min(C)
      (if g-min(C) ≥ 0, exit Simplex successfully)
2   Pivotcol = extract-column(A, pivotcolnum)
3   selecting positions in Pivotcol with values > 0
      (if none, exit Simplex unsuccessfully)
4   Ratio = p ÷ (B, Pivotcolumn)
5   pivotrownum = row # of element holding g-min(Ratio)
6   Pivotrow = extract-row(A, pivotrownum)
;update pivot row
7   pivotelement = A[pivotrownum][pivotcolnum]
8   Pivotrow = p ÷ (Pivotrow, Broadcast(pivotelement))
9   Rowmatrix = spread-row(Pivotrow)
10  Colmatrix = spread-column(Pivotcol)
;update constraint vector and objective function
;on their own, even though updated later
11  B = p - (B, p * (Pivotcolumn, Broadcast(pivotrow[m2])))
12  B[pivotrownum] = B[pivotrownum]/pivotelement
13  C = p - (C, p * (Pivotrow, Broadcast(C[pivotcolnum])))
;Update the tableau
14  A = deposit-row(A, Pivotrow, pivotrownum)
15  selecting positions of A that are not part of pivot row
16  A = p - (A, p * (Pivotrow, Pivotcolumn))

```

### 3.3 Implementation of Primitives

In this section we present efficient implementations for the four vector-matrix primitives on hypercube architectures and analyze the time and processor-time complexities of these implementations. The implementations are based on a particular embedding of matrices and vectors on the hypercube. We show that both the time complexity and the processor-time complexity of the *reduce* primitives are within a constant factor of optimal provided that the matrix is sufficiently large relative to the number of processors. Finally, we discuss generalizations to higher dimensional matrices, matrices whose dimensions are not powers of 2, and matrices where rows or columns are favored.

Figure 3-3 shows how we map an  $m_1 \times m_2$  matrix on an  $N$ -processor hypercube. Because we wish to optimize the worst-case performance of the primitives on any row or column, we treat each row and column as symmetrically as possible. Thus each processor holds a  $k_1 \times k_2$  submatrix which is as square as possible. The processors can themselves be viewed in a grid (perhaps in row-major order with respect to hypercube addresses or we can use Gray coding to map the grid onto the hypercube such that neighbors in the processor grid are joined by a hypercube wire). Our implementation, however, uses the full power of the hypercube connections. The processors holding a given row of the matrix form a  $\lg p_2$ -dimensional subcube of the  $n$ -dimensional hypercube and the given row is mapped to the same submatrix row in each processor. This is simply another way of saying that the  $\lg m_1$ -bit row address is the same for each element in a given row.

To map a length  $2^v$  vector onto an  $n$ -dimensional hypercube configured as a  $p_1 \times p_2$  grid, we map  $2^{v-n}$  vector elements to each processor with row vectors mapped in column-major order and column vectors in row-major order. Figure 3-5 illustrates how row vectors are mapped onto the virtual grid both when  $v > n$  and when  $v < n$ . The embedding of vectors as described above is *load balanced* in that each processor holds an equal number of elements.

Based upon the matrix and vector embeddings described above, we describe the implementation of the row version of the *extract* and *reduce* primitives. The implementation of the *deposit* and *distribute* primitives are similar and are pictured in Figures 3-7 and 3-10 with pseudocode given in figures 3-6 and 3-9. The *deposit* primitive is basically the inverse of the *extract* primitive. The *reduce* primitive is similar to the *extract* primitive but as well as swapping data across the cube to load balance, the data is summed along the way. The *distribute* primitive is basically the inverse of the *reduce* primitive.

All of the primitives are implemented by stepping through the dimensions of the hypercube with each processor simultaneously communicating with its neighbor in that dimension. In the *extract* and *reduce* primitives, the number of data elements communicated typically halves on each dimension step (each step halves the complexity). In the *deposit* and *distribute* primitives, the number of data elements communicated typically doubles on each dimension step. For the *extract* and *deposit* primitives, only one processor from each pair of communicating processors sends data. For the *reduce* and *distribute* primitives, both neighbors generally send equal amounts of data. None of the primitives require indirect addressing on the processors, and the control is strictly data parallel.

### 3.3.1 Extract

The **extract** operation takes a matrix and an index  $r$  within the bounds of the matrix and extracts row  $r$  as a row vector (see Figure 3-5). At the start of the algorithm, each processor either contains  $k_2$  elements of row  $r$  or it contains no elements of row  $r$ . Furthermore, those that contain elements of row  $r$  all have the same processor row address  $r_p$ , represented bitwise as  $r_0, r_1, \dots, r_{\lg p_1 - 1}$ . Thus neighbors across each of these dimensions do not contain any elements of row  $r$ . In the **extract** procedure, we step through the dimensions, starting at the highest dimension, and each processor containing elements of row  $r$  sends half of its elements to its empty neighbor in that dimension. The result of the **extract** is that the vector elements originally held in the processor at grid address  $(r_p, j)$  are evenly distributed among the processors in column  $j$ .

#### *Algorithm Extract-Row*

```

;matrix  $M$ 
;row number  $r$  (processor row  $r_p$ , offset  $r_m$ )

1  Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the bit representation
    of processor row  $r_p$ .
2  For  $i = 0$  to  $\lg p_1 - 1$  Do
3      Selecting processors with elements of row  $r$ 
4          If no processor has  $> 1$  element of row  $r$ 
5              then If  $r_i = 1$  then send value to neighbor in dimension  $i$ .
6          If  $r_i = 0$ 
7              then send last  $k_2/2^{i+1}$  elements of row  $r$  to neighbor in dimension  $i$ .
8              else send first  $k_2/2^{i+1}$  elements of row  $r$  to neighbor in dimension  $i$ .

```

The if statement in lines 6–8 guarantees that the row is kept in a fixed order. Figure 3-5a illustrates the case where the number of elements per row in each processor is greater than the number of processors in each column of the processor grid ( $k_2 \geq p_1$ ). In this case the row elements are dispersed in column-major order with each processor getting the same number of matrix elements. Figure 3-5b illustrates the case where the number of row elements per processor is less than the number of processors in a column of the processor grid ( $k_2 < p_1$ ). In this case, each processor has at most one matrix element. Only processors whose last  $\lg(p_1/k_2)$  bits are 0 contain vector elements because of the conditional sends in line 5 of algorithm **extract-row**. Thus the embedding of an extracted row is the same regardless of which row is extracted. This canonical representation of a vector is important when we wish to operate on two vectors such as to sum them or take a dot product. If, however, we only want to perform a global operation on the extracted vector and then throw it away, we can simply stop when each processor has at most one vector element.

It appears that we are using indirection in lines 7 and 8 of algorithm **extract-row**. In fact we are not. Since the value of  $r_i$  for  $0 \leq i < \lg p_1$  is the same in all processors, in each iteration of lines 2–8, exactly one of lines 7 and 8 is executed. We can copy the row to a working-space vector at the start of the algorithm at a cost of  $k_2$  and then each processor with elements of

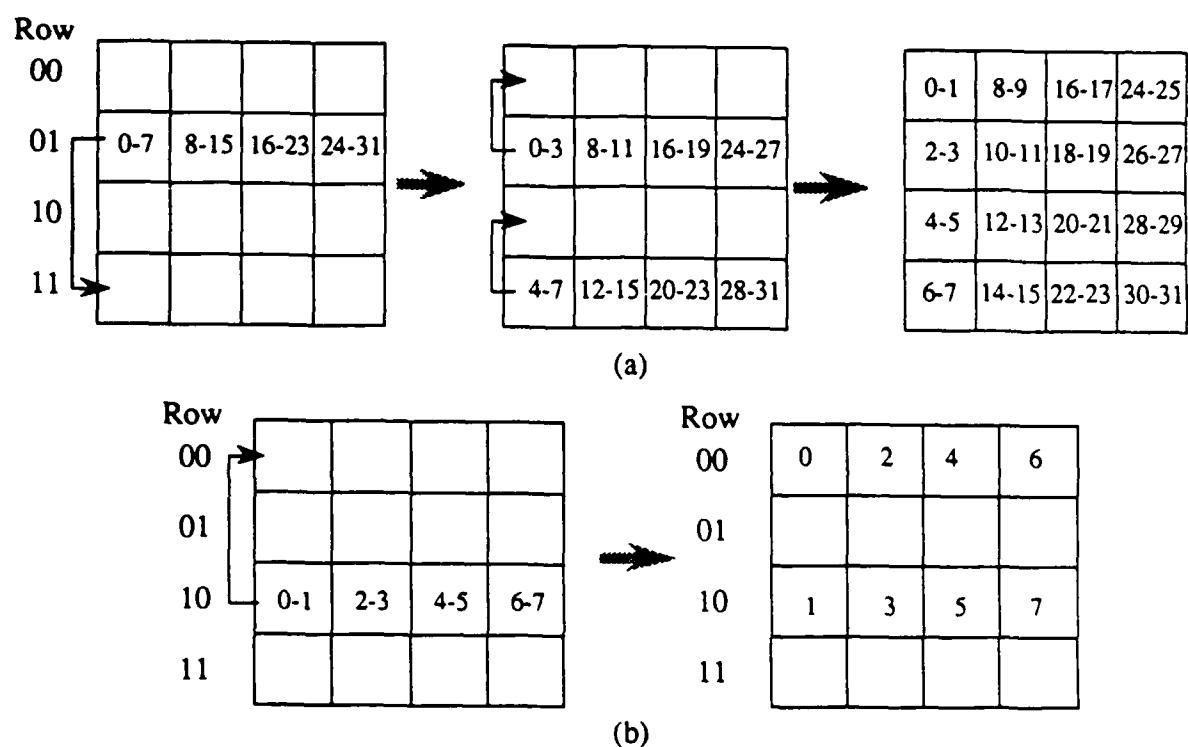


Figure 3-5: Examples of the algorithm *extract-row*. The grid indicates the grid of processors. Number ranges indicate elements of the row vector being extracted. At the start, all elements are in a single row of processors. In case (a), each processor ends up with more than one vector element. In case (b), some processor rows do not get any vector elements. In both cases, the final vector is in column-major order.

*Algorithm Deposit-Row*

```

;matrix M
;row number r (processor row  $r_p$ , offset  $r_m$ )

1 Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the bit representation of processor row  $r_p$ .
2 Let  $s_0, s_1, \dots, s_{\lg p_1 - 1}$  be the bit representation of a processor's own row address.
3 Let  $w$  be a local vector in each processor,
   initially holding all values of vector  $v$  in that processor.
4 For  $i = \lg p_1 - 1$  downto 0 Do
5   If  $s_i \neq r_i$  Then send all elements of vector  $w$  to neighbor in dimension  $i$ .
6   If processor received data
7     If  $s_i = 0$ 
8       Then store all elements received in the back of vector  $w$ .
9       Else store all elements received in the front of vector  $w$ .
10 Store vector  $w$  into local matrix row  $r_m$ .

```

Figure 3-6: Pseudocode for the deposit primitive. As with algorithm *extract-row* although it appears that we use indirection in lines 8 and 9 of algorithm *deposit-row* in fact we do not. All processors that receive data in line 5 have the same value of variable  $s_i$ , namely  $s_i = r_i$ .

row  $r$ , whether initially stored in the matrix or received during the course of the extraction, accesses the same memory location. Alternatively, we can perform the communication in two phases: one for the processors originally holding the row and one for all other processors, and then copy only  $\lceil k_2/p_1 \rceil$  vector elements at the end.

We now count the number of *messages* sent where a message is the sending of one matrix value by each active processor. During the first communication phase, each active processor sends  $k_2/2$  messages, then  $k_2/4$  messages, then  $k_2/8$  and so on. Each time a message is sent, each sending processor sends one element to a neighbor which has fewer elements. Thus, each message phase reduces the maximum number of elements per processor by one until the clean-up phase where at most one message is sent over the final  $\lg(\lceil p_1/k_2 \rceil)$  dimensions to place a vector in canonical form. Therefore, the total number of messages sent is equal to the reduction in maximum number of elements per processor plus clean-up costs. Since at most  $\lceil k_2/p_1 \rceil$  elements of  $r$  are left in any processor, the number of messages sent is  $k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)$ .

### 3.3.2 Reduce

For each binary associative operator  $\oplus$ , The  $\oplus$ -reduce-to-row operation takes an  $m_1 \times m_2$  matrix  $A$  and produces a row vector  $v$  such that  $v_i = a_{0,i} \oplus a_{1,i} \oplus \dots \oplus a_{(m_1-1),i}$ . For example, if  $\oplus$  represents addition, the  $i$ th element of the output vector contains the sum of all elements in matrix column  $i$ . At the start, each processor contains  $k_1$  elements of  $k_2$  different columns. After reducing within the processor, we are left with  $k_2$  different values: one partial sum from each of the  $k_2$  columns. We then step through each dimension of the processor row space, exchanging half of the elements (column sums) remaining in each processor, and computing on

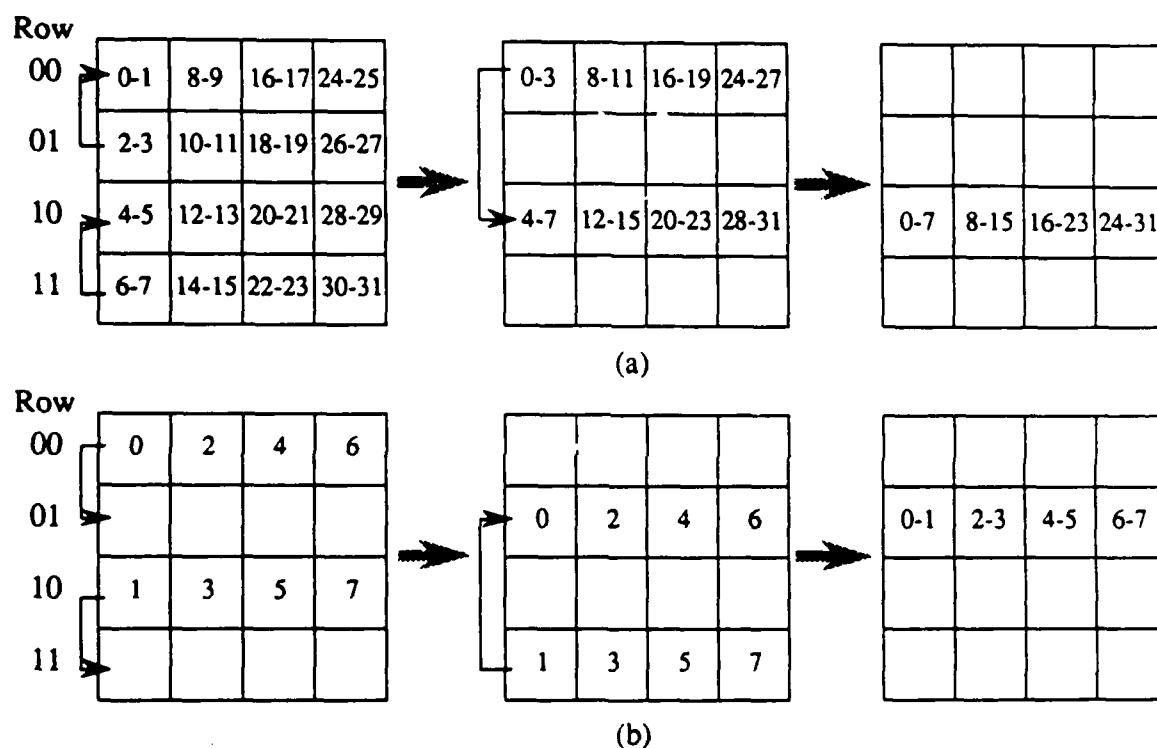


Figure 3-7: The deposit-row primitive deposits a row vector into a row of a matrix. In these examples, we deposit into row 2 (part a) and row 1 (part b). The algorithm is basically the inverse of the extract-row algorithm.

the remaining half.

In the following algorithm, we assume indirection so indexing into the working vector  $v$  is done via pointers which are updated after each communication phase.

*Algorithm  $\oplus$ -Reduce-to-Row*

*;matrix  $M$*

```

1  All processors Do in parallel
2    For each of the  $k_2$  columns within the processor Do
3      Compute the  $\oplus$  of the  $k_1$  elements in each column.
4      Store in vector  $v$ .
5  Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the bit representation of a processor's row address.
6  For  $i = 0$  to  $\lg p_1 - 1$  Do
7    if  $r_i = 0$ 
8      Then send last  $\lfloor k_2/2^{i+1} \rfloor$  elements of  $v$  to neighbor across dimension  $i$ .
9      Else send first  $\lceil k_2/2^{i+1} \rceil$  elements of  $v$  to neighbor across dimension  $i$ .
10   let  $w$  be the vector of values received from neighbor in dimension  $i$ .
11   for  $j = 0$  to  $k_2/2^{i+1}$  Do
12      $v_j = v_j \oplus w_j$ .
```

For the sake of discussion, let us assume the operator  $\oplus$  is addition. Figure 3-8 illustrates the *reduce-to-row* primitive operating on one column of processors. In figure 3-8a, the column of processors must produce the sum of 8 columns. Since there are only 4 processors in the column, each holds the final result of 2 columns. In figure 3-8b, there are fewer columns per processor than there are processors per column of the processor grid. Therefore, not every processor holds a column sum. For either case, the *reduce* operation produces a vector which is mapped onto processors in the canonical form described earlier (the same mapping produced by the *extract* operation, for example).

To modify the *reduce-to-row* routine to remove the need for indirect addressing, each processor permutes the partial sums in vector  $v$  as the intraprocessor reductions are computed. Vector  $v$  is permuted in such a way that during each step of algorithm *reduce-to-row* the processor sends the last half of vector  $v$ . More precisely, let  $b_0, b_1, \dots, b_{\lg k_2 - 1}$  be the bit representation of the index of an element in the unpermuted  $v$ . The index corresponds to intraprocessor column number of the partial sum. Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the processor row number. Element  $b_0, b_1, \dots, b_{\lg k_2 - 1}$  of the unpermuted vector  $v$  is stored in location  $b'_0, b'_1, \dots, b'_{\lg k_2 - 1}$  of vector  $v$  where  $b'_i = b_i$  if  $r_i = 0$  and  $b'_i = \bar{b}_i$  otherwise (including the case where there is no  $r_i$ ).

We now analyze the amount of work necessary to execute algorithm *reduce-to-row*. To reduce within a processor requires  $k_2(k_1 - 1)$  sums. We then have  $k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)$  messages each of which is followed by a sum. Therefore, we have at most  $k_2 k_1 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)$  sums and at most  $k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)$  messages.

Using arguments similar to those used in the above analyses, we see that *deposit-row* has the same complexity as *extract-row*. The message complexity of *distribute-row* is the same as *reduce-to-row*.

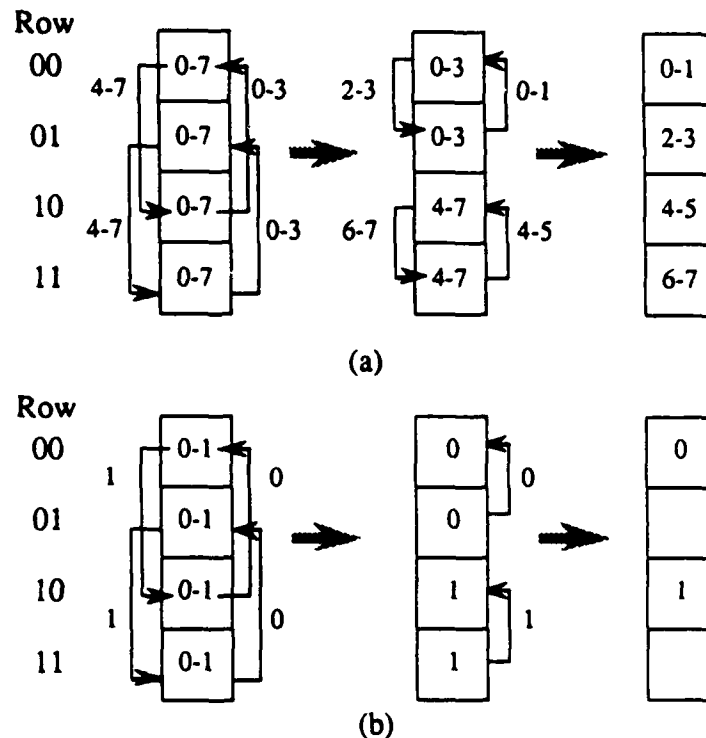


Figure 3-8: The reduce-to-row operation focusing on one column of processors. In example (a), we are reducing 7 columns. The ranges indicate that the processors contain the partial sums of the given column indices. In each step of the reduce-to-row operation, each processor sends half its elements to its neighbor in the  $i$ th dimension and accumulates what it receives into the half it keeps. The process terminates when all dimensions are processed. (a) If processors contain  $> 1$  final sum, then all processors have the same number of sums. (b) If all dimensions have not yet been crossed but processors have at most 1 partial sum, processors with higher addresses send their partial sum.

*Algorithm Distribute-Row*

;matrix  $M$  ( $m_1 \times m_2$ )  
 ;row-vector  $v$  (length  $m_2$ )

- 1 Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the bit representation of a processor's row address.
- 2 For  $i = \lg p_1 - 1$  down to 0 Do
- 3   send all elements of  $v$  to neighbor in dimension  $i$  (keep a copy).
- 4   If  $r_i = 0$
- 5     Then store vector elements received from neighbor at end of local copy of  $v$ .
- 6     Else store vector elements received from neighbor at front of local copy of  $v$ .

Figure 3-9: Pseudocode for the distribute-row operation. The distribute-row primitive takes an  $m_1 \times m_2$  matrix  $A$  and row vector  $v$  of length  $m_2$  and distributes a copy of that vector to each processor row. Logically, the distributed row is then replicated within each processor so that the result of the distribution is a matrix of size  $m_1 \times m_2$ . Practically, the processors need only keep one copy of the relevant piece of the row vector.

**3.3.3 Spreads**

Although we did not include it in our primitives, one could envision a **spread-row** operation which replicates a single row  $r$  of a matrix across each row of the matrix. A naive implementation of **spread-row** based upon the standard matrix embedding is as follows:

*Algorithm Spread-Row*

;matrix  $M$  ( $m_1 \times m_2$ )  
 ;and row number  $r$

- 1 Let  $r_0, r_1, \dots, r_{\lg p_1 - 1}$  be the bit representation of a processor's row address.
- 2 For  $i = 0$  to  $\lg p_1 - 1$  Do
- 3   send all elements of row  $r$  (if any) to neighbor in dimension  $i$  (keep a copy).

We could also define a **reduce-and-Spread** operation which reduces a matrix to a single row and then distributes that vector back across the matrix. A naive implementation of this operation would look very similar to the above algorithm except that both neighbors would communicate and sum all elements. The naive implementation of the **spread** operations requires  $k_2 \lg p_1$  messages.

We implement the **spread** and **reduce-and-Spread** operations directly from the **extract**, **reduce**, and **distribute** primitives. That is, to spread a row across a matrix, we first extract the row and then distribute the resulting row vector. To perform a **reduce-and-Spread** on rows, we first reduce to a row and then distribute the resulting column vector across the columns of the matrix. Somewhat surprisingly, it is much more efficient both theoretically and practically to first extract a vector (or reduce a matrix) than to operate on the matrix as a whole, even when

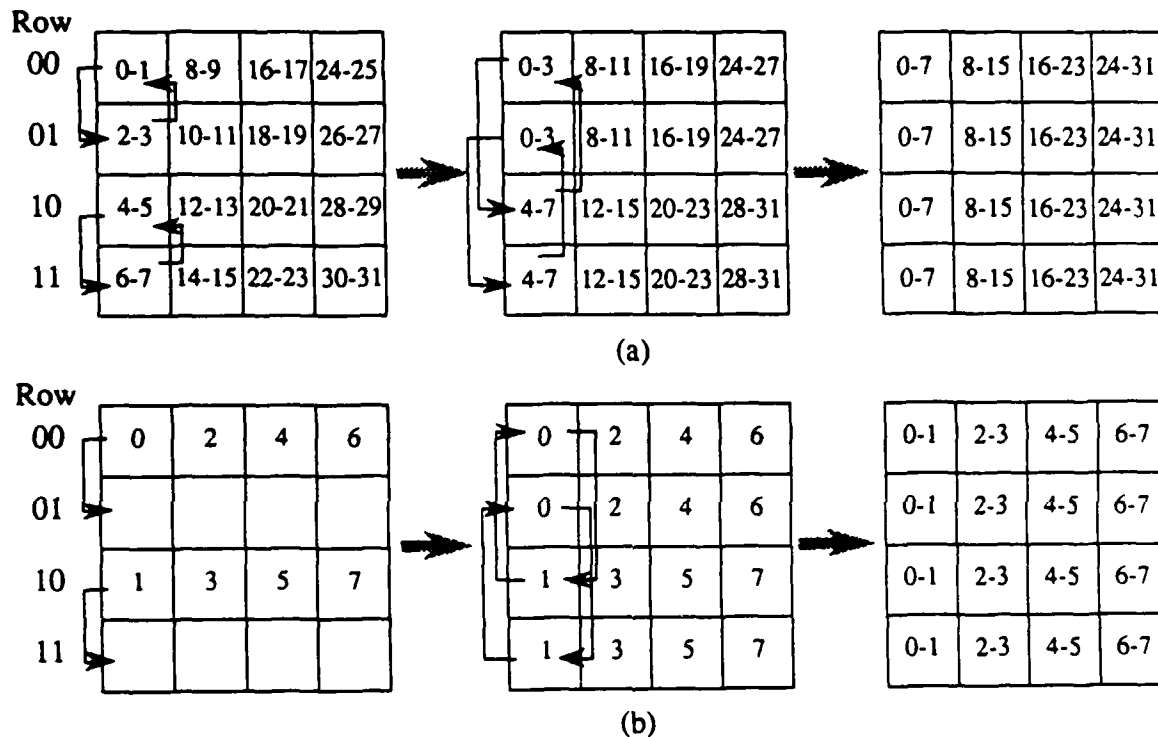


Figure 3-10: The distribute-row primitive replicates a row vector across a matrix. Communication proceeds across dimensions of the processor row address least significant bit to most significant bit. Each Processor sends all information to its neighbor and receives an equivalent amount of information, thus doubling data on each step. In case (b) where not all processors hold vector elements, the first steps replicate single values until all processors have one vector element. Then the number of values sent doubles on each successive step.

we are not interested in computing on the intermediate vector. The *spread-row* operation, if implemented with an *extract-row* followed by a *distribute-row* sends at most  $2(k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil))$  messages.

### 3.3.4 Analysis

In this section we argue that our algorithms for the *reduce-to-row* operations is within a constant factor of optimal in two ways. First we show that the parallel time (messages plus computation) required to execute algorithm *reduce-to-row* is within a constant factor of optimal. This is also true of the *distribute-row* implementation provided the embedding is nontrivial. Then we argue that the total work which is the product of parallel time ( $T$ ) and number of processors ( $P$ ) is within a constant factor of the work required for any sequential implementation of the *reduce* operation provided  $k_1 k_2 \geq \lg p_1$ . In other words, our algorithm is optimal if the number of matrix elements per processor is at least as large as the number of dimensions of the hypercube (in fact, a constant fraction smaller than  $\lg n$  is also sufficient). To argue optimality in terms of time, we state lower bounds for the number of messages and operations required by any parallel implementation of *reduce-to-row* and compare them to the operation counts argued earlier in this section. We then compare minimum number of operations with the  $PT$  product of our implementation of *reduce-to-row*. To achieve the lower bounds we use the *weak hypercube* model where each processor can only send out one message in unit time.

#### Parallel Operation Count

To prove that our implementation of *reduce-to-row* performs at most a constant factor more than the optimal number of parallel operations, we first prove that any parallel algorithm for *reduce-to-row* must send at least  $\Omega(\lg p)$  messages, where  $p$  is the number of processors, regardless of how the matrix elements are embedded in the hypercube. To complete the optimality proof for *reduce-to-row*, we finally show a lower bound of  $k_1 k_2$  parallel arithmetic operations on an  $N = p_1 p_2$  processor machine.

We now argue that any algorithm that computes *reduce-to-row* must send  $\Omega(\lg p)$  messages. Let us assume that we have  $m \geq p/2$ , because otherwise we could run our algorithm on a subcube of the machine. Also, let us assume without loss of generality that  $m_1 > m_2$ , so that  $m_1 > p/4$ . Suppose some processor contains  $\Omega(\lg p)$  elements from some column. Then it must accumulate at least half of them locally or send at least half of the elements to other processors, thus yielding the lower bound. Suppose, instead that no processor contains more than  $\lg p$  elements of any one column. Then some column must be embedded in at least  $m_1/\lg p$  processors. Therefore, to accumulate the column requires

$$\begin{aligned} \lg(m_1/\lg p) &= \lg m_1 - \lg \lg p \\ &\geq \lg(p/4) - \lg \lg p_1 \end{aligned}$$

messages. Therefore, we must send  $\Omega(\lg p)$  messages.

We now count the minimum number of parallel arithmetic steps on machine with  $N = p_1 p_2$  processors. The number of arithmetic operations that must be performed is  $m_2(m_1 - 1)$ . Since there are only  $p_1 p_2$  processors to perform that operation, we must use at least  $(m_1 m_2 - m_2)/p_1 p_2 = \Omega(k_1 k_2)$  time.

Combining the separate lower bounds, we have that the minimum time to perform a *reduce-to-row* is  $\Omega(\lg p + k_1 k_2)$ . Since the number of messages plus operations executed by our implementation of *reduce-to-row* is  $O(\lg p + k_1 k_2)$ , our implementation is optimal in terms of asymptotic parallel execution time.

We can use arguments similar to those above to show an  $\Omega(k_2 + \lg p)$  lower bound on message complexity for the *distribute* operation provided we require a processor holding  $e$  elements of a column to record  $e$  local copies of the distributed row element associated with that column. If the processors need only have one copy of the distributed row element for each column of which it contains any elements, then we can show the lower bound provided the matrix is embedded *nontrivially*. By *nontrivial*, we mean that each processor contains at least one matrix element and no one processor contains more than  $(1 - 1/p)m$  of the  $m$  matrix elements. We first show an  $\Omega(\lg n)$  bound for the number of messages needed to distribute a row or column provided each processor holds at least one matrix element. Then we show an  $\Omega(k)$  bound assuming a nontrivial matrix embedding for the above definition of nontrivial. We make no assumptions about the embedding of the vector to be distributed. The basic idea is that when a processor holds most of a column and the associated vector element that must be distributed to that column, then the row distribute for that column is trivial. *Column* distributes, however, for that processor then become costly.

We now show an  $\Omega(\lg p)$  lower bound on the time needed to distribute a row or column across a matrix even if we do not require local copying. Suppose that no processor contains more than  $\lg p$  elements of any row or column. Then each column is spread over at least  $m_1 / \lg p$  processors. Therefore the single value that must be distributed to all elements of a column must propagate to at least  $m_1 / \lg p$  processors. Using the same arguments as above, this propagation requires  $\lg(m_1 / \lg p) = \Omega(\lg p)$  messages.

Now suppose that some processor  $P'$  has at least  $\lg p$  elements from some column. Suppose that we now wish to distribute a column vector. If, on the one hand,  $\Omega(\lg p)$  of the necessary vector elements are stored in other processors, then we require  $\Omega(\lg p)$  time to get these values into the processor  $P'$ . If, on the other hand, processor  $P'$  holds  $\Omega(\lg p)$  of these vector elements then distributing within that processor is trivial since we do not require local copying. This processor must, however, send out any vector value it holds unless it, in fact, contains the entire row to which that value must be propagated. If it does not contain  $\Omega(\lg p)$  complete rows, therefore, we are done because we require  $\Omega(\lg p)$  time to ship out the vector values stored in processor  $P'$ .

Let us now consider the final case. We have a processor  $P'$  which contains at least  $\lg p$  elements from some column, holds  $\Omega(\lg p)$  of the vector elements for a column to be distributed, and contains  $\Omega(\lg p)$  complete rows (so there is no need to send out those values). What happens when a processor contains even one complete row? If we later wish to distribute a row, then this processor must receive all  $m_1$  values of the row. We achieve the desired lower bound based upon the time to receive these values unless processor  $P'$  holds at least  $m_1 - o(\lg p)$  of the vector elements. Under these circumstances, we achieve the desired lower bound based upon the time to send values unless processor  $P'$  holds at least  $m_1 - o(\lg p)$  complete columns. A similar argument shows that once a processor holds a complete column, then that processor must hold at least  $m_2 - o(\lg p)$  complete rows. Figure 3-11 demonstrates how many matrix elements processor  $P'$  must hold to avoid the lower bound. Thus there are at most  $o(\lg^2 p)$  matrix elements left to be embedded in the remaining  $p - 1$  processors. Since we assume that

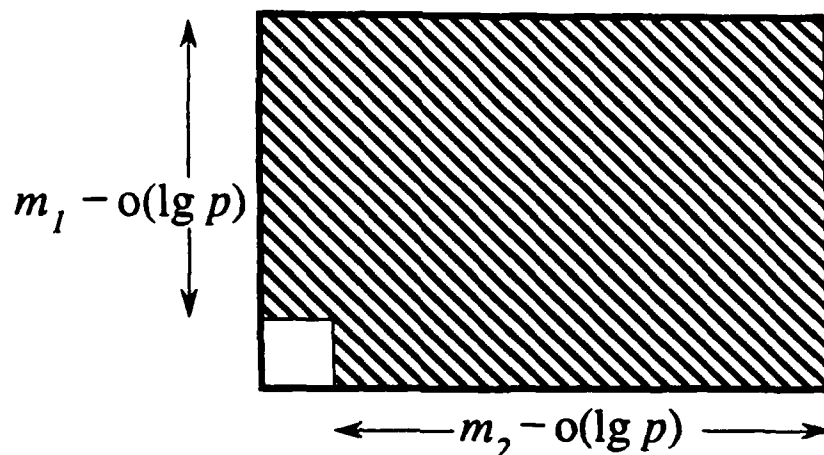


Figure 3-11: In the case where some processor  $P'$  contains at least  $\lg p$  elements from some column, we can show an  $\Omega(\lg p)$  lower bound on the time to send or receive messages from processor  $P'$  unless that processor contains all but  $o(\lg n)$  of the matrix elements. The hatched region represents the elements the processor must hold. The remaining  $o(\lg^2 p)$  elements cannot contain any full rows or columns.

each processor holds at least one matrix element, we have that  $p - 1 \geq \lg^2 p$  which is true asymptotically (for small values of  $p$ , in fact). Thus we have proven the  $\Omega(\lg n)$  lower bound.

We now prove an  $\Omega(k)$  lower bound on the worst-case time to distribute a row or column where  $k = \min k_1, k_2$ . Because we chose as square a submatrix as possible, we have that  $k^2 = m/p$  within a constant. Using arguments similar to the ones above, we see that we require  $\Omega(k)$  time unless some processor holds at least

$$\begin{aligned} m - k^2 &= m - m/p \\ &= m(1 - 1/p) \end{aligned}$$

elements of the matrix. This is impossible by our definition of nontrivial matrix embeddings.

We have shown an  $\Omega(k + \lg p)$  lower bound on the message complexity of distribute, even without local copy provided the matrix embedding is nontrivial. Since our implementation of the distribute primitives has message complexity  $O(k + \lg p)$  it is optimal within a constant factor.

### Processor-Time Product

In this section we show that the product of arithmetic operations times number of processors is within a constant factor of the number of operations required by any sequential reduce-to-row implementation. As argued above, any sequential algorithm must perform  $m_2(m_1 - 1)$  arithmetic operations.

The *PT* product for reduce-to-row is

$$PT = p_1 p_2 (k_1 k_2 + \lg(\lceil p_1/k_2 \rceil) - \lceil k_2/p_1 \rceil)$$

$$\begin{aligned}
&= m_1 m_2 + p_1 p_2 \lg([p_1/k_2]) - p_1 p_2 [k_2/p_1] \\
&\leq m_1 m_2 + p_1 p_2 \lg([p_1/k_2]) \\
&\leq m_1 m_2 + p_1 p_2 k_1 k_2
\end{aligned}$$

because we assume that  $k_1 k_2 \geq \lg p_1$ . Therefore we have that  $PT \leq 2m_1 m_2$ . This proves that the total work as indicated by the processor-time product is within a constant factor of optimal provided that  $k_1 k_2 \geq \lg p_1$ .

### 3.3.5 Computing on a Single Row or Column

In this subsection we discuss the cost of computing on a single extracted row of a matrix. In particular we consider whether it is always worth load balancing rather than just moving the row, or column, locally within the processors that contain it. We consider four cases that might appear in practice:

1. We extract a row, operate on it, and distribute it across the other rows.
2. We extract a row, operate on it, and deposit it back in place.
3. We extract a row, operate on it, and deposit it back into another row.
4. We extract a row, operate on it, and throw the row away (for example, if we wanted to find the maximum in a row).

The load balancing advantage costs nothing in the first case. Since spreading a row across to all others is efficiently implemented as an *extract* followed by a *distribute*, we simply break the spread into its two pieces and operate on the load-balanced representation in the middle. The decision of whether or not to load balance the vector in the second and fourth cases should be a compile-time or run-time decision, and depends on how many operations need to be performed, and the relative time of communication and computation. We have found in the applications we have studied that the first case occurs frequently, so it is therefore often worth load balancing when extracting a row. We analyze the second case to give an example of the considerations required for deciding whether it is worth load balancing or not.

In the second case we want to extract a row, operate on it, and put it back. Let  $a$  be the time it takes to perform a single arithmetic operation of interest (e.g. a divide) and let  $s$  be the time to send a matrix element over a hypercube wire. Let  $q = [k_2/p_1]$  be the maximum number of elements per processor after load balancing. Then the time it takes to compute without load balancing is  $k_2 a$ . If we extract first, compute and run *extract* in reverse, the cost of the entire computation is  $2(k_2 - q)s$  for the messages plus  $qa$  for the computation. The load balancing is advantageous exactly when

$$\begin{aligned}
k_2 a &> 2(k_2 - q)s + qa \\
(k_2 - q)a &> 2(k_2 - q)s \\
a &> 2s.
\end{aligned}$$

Thus a compiler need only estimate the amount of time to perform the arithmetic and the amount of time to send a matrix element over a cube wire. If the compiler determines that it is

not advantageous to do load balancing in such a situation, it can do a *lazy extract* which simply copies the row to a local array. If we want to store the result vector back to another row of the matrix, the decision is not entirely configuration independent since we must figure the cost of sending a row directly (worst case  $O(\lg p_1)$ ) vs. the cost of an extract plus deposit (worst case  $O(2(k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)))$ ), making load balancing less favorable. In this analysis we assumed there is no pipeline startup cost for executing multiple arithmetic or communication steps. With a pipeline start up cost, the decision could not be made at compile time.

### 3.3.6 Extensions

In this subsection, we discuss extensions to higher dimensional matrices, matrices whose dimensions are not powers of 2, and matrices where rows or columns are favored. We also discuss ways to represent vectors in a canonical form so that we no longer distinguish between row and column vectors. Finally, we discuss extraction of the main diagonals of matrices. Chapter 4 discusses an extension of the *extract* primitive that balances arbitrary distributions of active tasks.

If we have  $t$ -dimensional matrices, the address of each matrix element is now divided into  $t$  pieces corresponding to indices in each of the dimensions. All implementations extend directly by operating on the appropriate set of address bits. For example, to extract in dimensions 1 and 2, use the concatenated bits of the processor addresses for these dimensions in place of the row address in algorithm *extract-row*.

For matrices where  $m_1$  and  $m_2$  are not powers of 2, we embed the matrix in a  $p_1 \times p_2$  matrix such that each processor has at most  $k_1 = \lceil m_1/p_1 \rceil$  rows and  $k_2 = \lceil m_2/p_2 \rceil$  columns. If either  $k_1$  or  $k_2$  is not a power of 2, the first communication phase of an *extract* and the last communication phase of a *deposit*, etc, have fewer messages than normal, namely  $\lfloor k_i/2 \rfloor$ . Otherwise, the implementations are unchanged.

If we wish to optimize operations on rows, taking a penalty for operations on columns, we can configure the processor grid such each processor holds a minimum number of elements from the same row (at most  $\lceil m_2/p \rceil$ ).

Another possible extension is to store vectors in a canonical form. In our current implementation a vector extracted from a row of a matrix (a row vector) is ordered on the processors differently from a vector extracted from a column of a matrix (a column vector). It is possible to store all vectors in the same ordering, a canonical form, with no additional cost to our primitives. This, however, requires that the rows and columns of a matrix are mapped onto the processors in a noncontiguous order—one row of a matrix is no longer adjacent to the next. This makes nearest neighbor communication on a grid more expensive. We believe that the better choice is to keep the two representations and swap between them when necessary. This could be hidden from the user.

The *extract* primitive can be applied to the main diagonal of a matrix without modification. If we have square submatrices stored in each processor ( $k_1 = k_2$ ), then we have only one processor in each row of processors that contains any active matrix elements. We can then extract this diagonal by proceeding through the dimensions of the processor column address as before. The only difference is that the memory location of the active elements may not be the same for every processor. This procedure can be used with either the rows or the columns (but not necessarily both) of the processor grid regardless of the aspect ratio of the matrix blocks

within each processor.

### 3.4 Timings

We present the timings for each of the primitives on the CM-2 along with the timings for the vector matrix multiply and Simplex algorithms. The timings are for a square matrix in a 16384 processor machine running at 6.7 MHz. Timings reflect the total time that the CM-2 was busy. Each element of the matrix is a single precision (32 bit) floating point number. Timings for the full 65,536 processor machine are extrapolated from the figures for the smaller machine. We give the timings for different values of  $k_1$  and  $k_2$ , the intraprocessor matrix dimensions. The times for more than 1 element per processor scale sublinearly with the total number of elements in each processor. The times are presented in milliseconds, and the flop rate in Mflops.

	Reduce			
Elements per proc	16K		64K	
	msec	Mflop	msec	Mflop
1 × 1	0.70	23	0.86	75
2 × 2	0.98	66	1.14	229
4 × 4	1.75	149	1.91	558
8 × 8	4.30	243	4.46	939
16 × 16	13.27	315	13.46	1246

	Distribute			
Elements per proc	16K		64K	
	msec	Mflop	msec	Mflop
1 × 1	0.44	-	0.44	-
2 × 2	0.69	-	0.69	-
4 × 4	1.30	-	1.30	-
8 × 8	3.09	-	3.09	-
16 × 16	8.81	-	8.92	-

	Extract			
Elements per proc	16K		64K	
	msec	Mflop	msec	Mflop
1 × 1	0.69	-	0.85	-
2 × 2	0.88	-	1.04	-
4 × 4	1.21	-	1.37	-
8 × 8	1.95	-	2.11	-
16 × 16	3.46	-	3.65	-

	Vector-Mtx Mpy			
Elements	16K		64K	
per proc	msec	Mflop	msec	Mflop
1 × 1	1.87	17	2.19	59
2 × 2	2.67	48	2.99	174
4 × 4	4.37	110	5.05	415
8 × 8	11.19	187	11.51	728
16 × 16	32.90	254	33.39	1004

We now provide an example of performance improvement from use of these primitives. A carefully coded naive implementation of Simplex, using  $k_1 = 8$  and  $k_2 = 8$  ran at a speed of 190 milliseconds per iteration, which is equivalent to 44 Mflops. The version using the primitives discussed in this paper takes 17 ms per iteration, which is equivalent to about 500 Mflops.

The distribute-row implementation used for the matrix-vector multiply timing propagated row values through the intraprocessor matrix. Performance can be improved by propagating only one copy of each value per processor. We estimate the time spent distributing the values across the intraprocessor matrix to be as high as 85% of the overhead for  $16 \times 16$  intraprocessor matrices. Hence, we could gain considerable performance by implementing this optimization.

### 3.5 Conclusions

In this section we offer additional comments on the nature of the extract implementation and point to future research directions.

Abstractly, each exchange of matrix elements across a dimension takes a bit that is constant in the processor addresses of active processors, making it vary, and takes a bit that varies in the intraprocessor address of the active processors, making it constant. That is, we exchange a constant in processor space for a constant in intraprocessor space. For example, in figure 3-5(a), we are extracting a matrix row that is embedded in processor row 01. At the start of the extraction, the four active processors all have processor row 01. Each active processor contains 8 elements of row  $r$ , so to represent the intraprocessor offset requires 3 bits. After the first communication phase, active processors have row number 01 or 11, so the first bit is no longer constant. Each active processor now holds 4 elements of row  $r$ , so to represent the intraprocessor offset requires only 2 bits. Therefore the leading bit of the intraprocessor address is now a constant (0). Alternatively, we can think of the matrix as being embedded in a full  $(\lg m_1 + \lg m_2)$ -dimensional hypercube. Of course, in this case dimensions are not equivalent since some are across processors and some across processor memory. Algorithm extract-row rotates the hypercube such that a given row has a maximum number of its dimensions across processor space.

In the future we hope to generalize our implementation of the primitives so they work on processor grids whose row and column sizes are not powers of two, and to allow a vector extracted from a row of a matrix to be distributed to or deposited in either a row or column of another matrix. We also hope to make our primitives available to higher level languages, as part of a library of related matrix primitives, so that they can be easily used.

We hope that this chapter spurs interest in developing a small set of simple matrix primitives which could then be efficiently implemented with a consistent interface on a large number

of machines. The primitives described here are not parallel primitives. They are equally meaningful on a sequential architecture. It is our *implementation* that is parallel.

## Chapter 4

# Dimension-Exchange Load Balancing

This chapter considers a load-balancing technique for hypercube multiprocessors that is a generalization of the extract primitive presented in chapter 3. When we compute upon a row or column of a matrix embedded as described in section 3.1, we know exactly where the active elements are. In general, however, each processor can have an arbitrary number of computations to perform on any given instruction. In a data-parallel architecture, we cannot begin a new instruction until all processors have completed the previous instruction. Thus, our computation speed is limited by the processor holding the maximum number of active jobs on each instruction. *In general, the maximum can be much larger than the average load, so it may be advantageous to balance the load among all processors.* Since all processors are performing the same operation, we can send a task to another processor by simply sending the data to be operated upon. Thus the notions of tasks and data elements are interchangeable in this context.

It is possible to balance load exactly (within 1) using global information. One possible way is to use an extension of an algorithm developed to contract holes out of lists on the Connection Machine. Suppose there are  $W$  tasks spread arbitrarily across the  $n$  processors. We enumerate the tasks using a parallel prefix computation and then send the first  $\lfloor W/n \rfloor$  tasks to processor 0, the second  $\lfloor W/n \rfloor$  to processor 1, and so on. If we have  $W = m \bmod n$ , then the first  $m$  processors receive an extra task. More precisely, if processor  $i$  has  $w_i$  tasks, then after the parallel prefix computation, each processor knows the number of tasks held by processors preceding it in linear order:  $W_i = \sum_{j < i} w_j$ . Thus each processor can compute to which block(s) of size  $W/n$  its tasks belong, and send the tasks to the appropriate processor(s).

This global load-balancing strategy has some disadvantages. First of all, in general a processor may have to ship all its tasks off to a distant processor and then receive an equal or greater number of tasks from some other distant processor. Thus we lose much locality. Secondly, each message must travel more than one hypercube wire in general, and therefore this strategy requires a router. Because each processor sends and receives many messages, the router is likely to take a long time to route all the messages, especially if the machine does not have hardware to assist in handling collisions. We would prefer to have each processor send (receive) only as many messages as necessary to give it the average load. Computing this message pattern, however, would require much more work than a single parallel prefix computation followed by

a local computation.

In this section, we investigate a strategy for general load balancing that uses only local information. To determine where to send each task, the above global strategy performs a parallel prefix computation which requires  $\Theta(\lg n)$  time<sup>1</sup> on an  $n$ -processor machine. The local strategy uses  $\Theta(\lg n)$  very simple one-wire communications.

For any distribution of computations on a hypercube, we can balance the load by proceeding through each dimension in turn and balancing the load between neighbors in that dimension. Thus during each communication phase, which we call an *exchange*, a subcube of half the machine size is communicating to the rest of the machine which is also a subcube of half the machine size. Logically, data is transferred first across halves of the machine, then simultaneously across halves of two half-sized subcubes, and so on. This type of communication is sometimes called *cube swapping*.

If tasks can be divided infinitely, then one *pass*, consisting of an exchange across each of the  $\lg n$  dimensions in turn, distributes work perfectly. Thus if processor  $i$  has  $w_i$  tasks, then after balancing, each of the  $n$  processors has exactly the average number of tasks:  $(\sum_i w_i)/n$ . To see this, suppose that exactly one processor  $p$  has  $w$  tasks and all other processors have none. After processor  $p$  exchanges across dimension 1 with its 1st-dimensional neighbor, the two processors have  $w/2$  tasks. After exchanges across the first  $i$  dimensions, all  $2^i$  processors in the subcube with identifiers matching  $p$  in the last  $\lg n - i$  bits have exactly  $w/2^i$  tasks. After processing all dimensions, all processors have exactly  $w/n$  tasks. Thus the work from processor  $p$  has been exactly balanced across the whole machine. Using superposition, the work from each processor is exactly balanced across the whole machine and thus each processor ends up with the average number of tasks. Of course, in practice, instead of having each processor send half its work to its neighbor on each exchange, as implied by superposition, the two neighbors compare work and send tasks from the more heavily loaded to the less heavily loaded. Interestingly, if we exclude even one dimension, the processors can be balanced arbitrarily badly.

Cybenko [32] calls this technique the *dimension-exchange* method and he proves that it is superior to a *diffusion* method where each processor averages the work among all neighbors on each step. He expresses the work load as a vector of length  $n$  and analyses the matrix that appears in the recurrence relation for iteratively updating the work vector. For example, he formally proves using these matrices that the dimension-exchange method provides perfect balancing. The optimal linear diffusion method, however, only brings the work vector  $e^2$  times closer to the uniform distribution after  $\lg n$  iterations, where distance between work vectors is the standard Euclidean norm. Thus the diffusion method does not provide as good a balance as the dimension-exchange method even though it requires more work. Each processor communicates with all  $\lg n$  neighbors during each of the  $\lg n$  iterations of the diffusion method, while each processor communicates with only one neighbor during each of the  $\lg n$  exchanges.

Cybenko also proves that the dimension-exchange method is superior to the diffusion method if after each exchange some work is completed and more work is independently randomly generated. Both methods converge toward the uniform distribution, but the dimension-exchange method has a variance that is a  $\lg n$  factor smaller than the diffusion method. This result is a further justification of the use of the dimension-exchange method in the implementation of the

<sup>1</sup>This assumes the EREW PRAM model. In the scan model, a parallel prefix is counted as a unit-time primitive.

vector-matrix primitives discussed earlier and it motivates further investigation of the method. We will not, however, consider the case of randomly generated additional work because we are primarily interested in data-parallel architectures. In such architectures, when each processor is simulating the work of many others, the tasks to be computed in each round are independent. Therefore it doesn't matter which processor completes which task, and we can consider load balancing. Determining the active tasks for the next round, however, generally requires information from the preceding computation and/or information from the original processor. Furthermore, because in a data-parallel architecture all processors execute the same instruction on any given cycle, we cannot start computing tasks for the next round unless that round requires the same operation as the current one.

Cybenko's analysis allows tasks to be infinitely divisible. In this section, we consider the case where computations are indivisible, as they are in practice when the tasks are primitive operations. In this case, after processing dimension  $i$ ,  $i$ th-dimensional neighbors have an equal number of tasks to within one. We examine strategies for distribution of the extra task in the case where the sum of tasks held by two neighbors is odd. We define the *excess* of a processor to be the difference between the number of tasks actually held at some point in the balancing process and the number that processor would hold if tasks were infinitely divisible. The excess of a subcube is the sum of the excesses of the processors in the subcube. By this definition, excess can be negative which implies a deficit. We analyze the worst-case excess of any processor. In particular we show that after a single pass, the processor with the most elements has an excess of  $\Theta(\lg n)$ . That is, it has  $\Theta(\lg n)$  elements over the average, where  $n$  is the number of processors in the hypercube.

In section 4.0.1 we show that we can achieve  $\lg n/2$  as an upper bound on the worst-case excess using any strategy for dealing with extra tasks. If we choose to send extra tasks to the processor whose identifier has even parity, (i.e., an even number of 1's in the bit representation), then we achieve  $\lg n/4$  as an upper bound to within a small additive constant.

In section 4.0.2 we show that for any strategy for distributing extra tasks, there exists some distribution of work such that after load balancing, the worst-case processor has an excess of  $\Omega(\lg n)$ . We present a lower-bound proof which suffices for any deterministic oblivious dimension-exchange strategy that divides work evenly (within one). The term *oblivious* in the routing context means that processors make strictly local decisions, with no global knowledge of message distribution. In this context, we mean that not only do two processors use no global information when dividing work, but also they use no information about earlier phases of the load balancing. We explain why we do not feel that the latter is a serious limitation.

Finally, in section 4.0.3 we consider the benefits of reduced load vs. the added message complexity of the load-reduction process, an issue that Cybenko does not address. In particular we show that there is a distribution of work which requires  $\Omega(w \lg n)$  messages to achieve a reduction in worst-case load of  $O(w)$ .

#### 4.0.1 Upper Bounds

In this section we analyze the worst possible excess any processor can have after one pass of dimension-exchange load balancing. We assume that exchanges divide work evenly (within 1) among the communicating neighbors, and consider the problem of choosing how to distribute the extra task when there is one. We show that no strategy for distributing extra tasks can

yield an excess greater than  $\lg n/2$  in any processor. We also show that by choosing to give any extra tasks to the neighbor whose identifier has even parity, no processor can have an excess greater than  $\lg n/4$  to within an additive constant.

We begin by defining some terminology used to ease the exposition. We call a communicating pair of processors *even* (*odd*) if the sum of the tasks held by the two processors is even (odd). We define the *level- $k$  subcubes* of an  $n$ -processor hypercube, for  $k = 1, 2, \dots, \lg n$ , as the  $2^k$  subcubes of size  $n/2^k$  formed by grouping processors whose identifiers match in the first  $k$  bits. We use the  $k$ -bit prefix to distinguish among the level- $k$  subcubes. Thus, for example, the "11-subcube" is a level-2 subcube containing all the processors whose identifiers have 11 as the first two bits. In the following discussion, we sometimes refer to the bits in the bit-representation of a processor's identifier as simply the "processor's bits". Since we are concerned only with the number of units of data transmitted, and have no reason to examine the individual bits of data, the terminology should not cause confusion.

We now look at the effect of one exchange upon a pair of communicating processors. Suppose processors  $P_1$  and  $P_2$  are  $i$ th-dimensional neighbors and suppose that before the  $i$ th exchange, processor  $P_j$  holds  $w_j$  tasks giving it an excess of  $e_j$  for  $j = 1, 2$ . If the pair is even, then after the  $i$ th exchange, processors  $P_1$  and  $P_2$  each hold exactly  $(w_1 + w_2)/2$  tasks and each has an excess of  $(e_1 + e_2)/2$ . Thus even pairs never create excess beyond what is already there. If the pair is odd, however, then after the exchange, one processor holds  $\lceil (w_1 + w_2)/2 \rceil$  tasks, giving it an excess of  $(e_1 + e_2 + 1)/2$ , and the other holds  $\lfloor (w_1 + w_2)/2 \rfloor$  tasks, giving it an excess of  $(e_1 + e_2 - 1)/2$ .

We now consider the load-balancing process in terms of higher-level cube swapping. When performing the dimension-exchange load balancing on an  $n$ -processor hypercube, the first exchange causes a flow of data between the two  $n/2$ -processor level-1 subcubes. One can think of a plane cutting the hypercube in half. The  $n/2$  wires that cross the plane form the communication *boundary*. We say the boundary has size  $n/2$ . Processors with leading bit 1 are on one side, and those with leading bit 0 are on the other. After this exchange, no additional communication occurs between these two subcubes. Because all remaining exchanges move data within these subcubes, the excess each subcube holds at the end is exactly the excess obtained during the first exchange.

We can think of the two subcubes as being disconnected after the exchange, and we can apply the above arguments recursively. During the  $i$ th exchange, we once again have  $n/2$  processor pairs communicating, but logically data now flows between  $2^{i-1}$  pairs of level- $i$  subcubes. Two subcubes communicate during the  $i$ th exchange if they contain processors whose identifiers match in the first  $i - 1$  bits. The subcubes communicate across a boundary of size  $n/2^i$  — those with  $i$ th bit 1 on one side, and those with  $i$ th bit 0 on the other. Once again, any excess sent to one side or the other is irreparably stuck within the given subcube for the remainder of the load balancing process. The final exchange is logically among  $n/2$  "disconnected" pairs.

From the above discussion, we see that for a communication between two subcubes, the worst-case excess occurs when each pair of processors on the boundary is odd and each sends its excess task to the same side of the boundary. We now show that even if the worst-case balancing occurs at each exchange, no processor has an excess of more than  $\lg n/2$  after the  $\lg n$ th (final) exchange.

Let  $E(k)$  be the maximum excess of any level- $k$  subcube after the  $k$ th exchange. The worst case on the first exchange occurs when all  $n/2$  pairs of communicating processors are odd and

all send their extra task to side 1, without loss of generality. In this case, since each side should have received half of the extra  $n/2$  to achieve perfect balance, side 1 has an excess of  $E(1) = n/4$  and side 0 has an excess of  $-n/4$ . Next we focus on the 1-subcube and consider the exchange across the 2nd dimension. If we once again assume the worst case, then all of the  $n/4$  processor pairs on the boundary between the appropriate level-2 subcubes are odd and all send their extra to side 1, (i.e., to the subcube with leading bits 11), without loss of generality. The 1-subcube had an excess of  $n/4$  after the first exchange, so each of its level-2 subcubes inherits half of that excess, or  $n/8$ . In addition, the bad split on the second exchange gives the 11-subcube an excess of half the boundary size, or  $n/8$ , using the above arguments. Thus we have  $E(2) = n/8 + n/8 = n/4$ . In general we have the following recurrence, if the worst case occurs at each level:

$$\begin{aligned} E(k) &= \frac{E(k-1)}{2} + \frac{n}{2^{k+1}} \\ E(0) &= 0. \end{aligned}$$

The second term of the sum in the first equation is equal to half the size of the boundary between subcubes communicating during the  $(k+1)$ st exchange. Because excess is defined as a rational quantity, the recurrence is exact.

We now solve the recurrence to show that no processor has excess greater than  $\lg n/2$ .

**Lemma 7**  $E(\lg n) = \lg n/2$ .

*Proof.* We begin by showing that the solution to the recurrence is

$$E(k) = \frac{kn}{2^{k+1}} \quad (4.1)$$

and then substitute  $k = \lg n$  to achieve the desired upper bound.

We prove equation 4.1 by induction. Substituting  $k = 0$  into equation 4.1, we have that  $E(0) = 0$  as required.

Assume that for all  $j < k$  we have  $E(j) = jn/2^{j+1}$ . We have from the recurrence relation that

$$E(k) = \frac{E(k-1)}{2} + \frac{n}{2^{k+1}}.$$

Substituting for  $E(k-1)$  using the induction hypothesis we have

$$\begin{aligned} E(k) &= \frac{(k-1)n}{2^{k+1}} + \frac{n}{2^{k+1}} \\ &= \frac{kn}{2^{k+1}} \end{aligned}$$

which completes the induction.

Substituting  $k = \lg n$  into equation 4.1 yields

$$\begin{aligned} E(\lg n) &= \frac{n \lg n}{2^{\lg n + 1}} \\ &= \frac{n \lg n}{2n} \\ &= \lg n/2 \end{aligned}$$

which completes the proof. ■

We now analyze a specific strategy for assigning extra tasks: whenever a pair of communicating processors is odd, assign the extra task to the processor whose identifier has even parity. By the definition of a hypercube given in section 1.2, two processors are neighbors if and only if the bit representation of their identifiers differs in exactly one bit. Therefore, for any pair of neighboring processors, one has an identifier with even parity (an even number of 1's), and the other has an identifier with odd parity (an odd number of 1's). Furthermore, a hypercube with  $n$  processors has  $n/2$  processors with even parity and  $n/2$  processors with odd parity. Therefore, when we look at all the pairs communicating across the boundary between two subcubes, half the pairs will send any excess to the 1 side and half the pairs will send any excess to the 0 side.

Let us consider the worst-case exchange among two subcubes using this strategy for assigning extra tasks. It is not the same configuration as above because if all processor pairs on the boundary are odd, then half the extra tasks will go to each side, yielding a perfect split for this level. The worst case is achieved when all pairs which send excess to side 1 are odd and all pairs which send excess to side 0 are even. After the first exchange, the worst-case level-1 subcube has an excess of  $n/8$ , since it received  $n/4$  more tasks than the other. Using the same arguments as we used above we see that the recurrence is exactly the same except that the excess added on each level is half what it was before:

$$\begin{aligned} E(k) &= \frac{E(k-1)}{2} + \frac{n}{2^{k+2}} \\ E(0) &= 0. \end{aligned}$$

The recurrence is valid only through the first  $\lg n - 1$  exchanges since on the last ( $\lg n$ th) exchange, the boundary is less than 2. Thus for completeness' sake we add

$$E(\lg n) = E(\lg n - 1)/2 + 1/2.$$

Using the same arguments as used in the proof of lemma 7, we see that  $E(k) = kn/2^{k+2}$ . Substituting  $k = \lg n - 1$  yields  $E(\lg n - 1) = \lg n/2 - 1/2$ , and substituting this into the final clause of the recurrence yields  $E(\lg n) = \lg n/4 + 1/4$ . Therefore this strategy never leaves any processor with more than  $\lg n/4 + 1/4$  excess regardless of the initial work distribution.

#### 4.0.2 Lower Bounds

In this section we show that for any strategy for distributing extra tasks, there exists some distribution of work such that after load balancing, the worst-case processor has an excess of  $\Omega(\lg n)$ . We present a lower-bound proof which suffices for any deterministic oblivious dimension-exchange strategy that divides work evenly (within one). By "oblivious" we mean that when assigning extra tasks, two processors use no global information about the work distribution and no information about earlier phases of the load balancing.

We place the first restriction because the dimension-exchange method is inherently a local strategy, and comparing its balancing performance against that of a global strategy is unfair unless one considers all the costs involved. If processors have full global knowledge, then they can simply ship the data to the optimal location, but the cost of acquiring that knowledge is substantial. As explained in the introduction to this section, limiting an algorithm to local information greatly simplifies its communication pattern. For example, we do not need a

router or the complicated wire manipulations required for a parallel prefix computation. For motivation's sake, one can assume that we are operating on a machine without a router, for example.

The second restriction — no information about earlier phases of the computation — is not a serious limitation. As we explained in section 4.0.1, once two subcubes have exchanged information and data, they are essentially disconnected for the remainder of the algorithm. Thus any information acquired by a processor  $p$  during earlier exchanges is pertinent only to processors from which  $p$  is now disconnected, and it cannot affect the quality of any future balancing decisions. This information can at best act as a random bit, and therefore it cannot improve the worst-case scenario for any task-division strategy.

The algorithms we consider can be thought of as functions of the form  $A(p_1, p_2, w_1, w_2, i)$ , where the  $p_j$  are processor identifiers, and  $w_j$  is the number of tasks in processor  $p_j$  before the  $i$ th exchange. As explained in section 4.0.1, whenever a communicating processor pair is even, the number of tasks is divided perfectly. We can think of algorithm  $A$  as an *arbitrator* which is called upon to assign extra tasks when necessary. Therefore, the function  $A$  is used only in the case where  $p_1$  and  $p_2$  are  $i$ th-dimensional neighbors and  $w_1 + w_2$  is odd. For these cases, algorithm  $A$  should output a 1 or a 0, indicating the extra task should go to the processor whose  $i$ th bit is 1 or 0 respectively. This corresponds to indicating one of the communicating level- $i$  subcubes.

We now show that for any such algorithm  $A$ , there exists an initial distribution of tasks in the  $n$ -processor hypercube such that after dimension-exchange load balancing with algorithm  $A$  as arbitrator, some processor has excess  $\Omega(\lg n)$ . The idea is to cause a bad split on exchange  $i$  within the worst-case level- $(i-1)$  subcube. One of its two level- $i$  subcubes acquires an additional excess proportional to the boundary of the communicating subcubes, and thus proportional to the size of the subcube itself. More precisely, two subcubes of size  $b(i) = n/2^i$  communicate across a boundary of size  $b(i)$  and one of them acquires an additional excess of  $b(i)/16$ . We can express the final excess in the worst-case processor by solving a recurrence similar to those solved in section 4.0.1.

We find the bad initial distribution by working backwards. The *input* to a level- $i$  subcube is a vector indicating the number of tasks held by each of its processors after the  $i$ th exchange (i.e., just before its two halves communicate). For each level- $i$  subcube, we find constraints on its inputs which force bad splitting on levels  $i+1, i+2, \dots, \lg n - 3$  within that subcube. That is, if the constraints are met at level- $i$ , then one of the level- $(i+1)$  subcubes  $H$  will receive excess proportional to  $b(i+1)$ , one of subcube  $H$ 's level- $(i+2)$  subcubes will receive excess proportional to  $b(i+2)$ , and so on until a "large" amount of excess is driven into a constant-size subcube.

After forming constraints for all level- $i$  subcubes, we then pair the level- $i$  subcubes which communicate on the  $i$ th exchange. Two communicating level- $i$  subcubes form the 1 side and 0 side of a level- $(i-1)$  subcube. Then we find constraints on the inputs to each level- $(i-1)$  subcube such that

1.  $1/8$ th of the processor pairs that communicate on the  $i$ th exchange are odd and favor the same side (0 or 1),
2. the rest of the pairs are even, and
3. the input constraints of the level- $i$  subcube that acquires the excess are met.

These new constraints on the level- $(i + 1)$  subcubes, if met, now force bad splits on levels  $i, i + 1, \dots, \lg n - 3$ .

We now describe how to form the constraints. There are three types. The first is of the form  $x_{p,i} = k$ , where  $x_{p,i}$  denotes the input to processor  $p$  (number of tasks) before the  $i$ th exchange and  $k$  is a fixed value. These constraints, called *force constraints*, are used to force the inputs to pairs that we wish to be odd favoring a certain side. The second type of constraint is of the form  $x_{p,i} + y_{p,i} = k$ , where  $y_{p,i}$  denotes the input to processor  $p$ 's  $i$ th-dimensional neighbor before the  $i$ th exchange. Thus if processor  $p'$  is the  $i$ th-dimensional neighbor of processor  $p$ , then we have  $x_{p',i} = y_{p,i}$  by definition. These constraints, called *sum constraints*, are used to meet force constraints on level  $i + 1$ . The third type of constraint is of the form  $x_{p,i} \geq i$ , and there is such a constraint for all  $p$  and all  $i$ . We add these *min-size constraints* as a convenience. Without them, when we choose values for the variables in sum constraints, if one of the addends, say  $x_{p,i}$ , is very small, then we are extremely limited in our choices for inputs to processor  $p$  in the exchanges preceding the  $i$ th. We meet the min-size constraints at each level by careful construction of the force and sum constraints.

We begin our search for a bad initial distribution, for some algorithm  $A$ , by choosing constraints on the inputs to each of the  $n/16$  level- $(\lg n - 4)$  subcubes of size 16. On the  $(\lg n - 3)$ rd exchange, we have subcubes of size  $b(\lg n - 3) = 8$  communicating: the 1 side and 0 side of each level- $(\lg n - 4)$  subcube. Looking at one such subcube, we pick any one pair of communicating processors, say processor  $p$  and its neighbor, to be odd. We then add the constraints

$$\begin{aligned} x_{p,\lg n-3} &= \lg n - 3 \\ y_{p,\lg n-3} &= \lg n - 2. \end{aligned}$$

These choices for inputs makes the pair odd and meets the min-size constraints. All remaining pairs must be even, so for all other communicating pairs in the subcube, we add the constraint

$$x_{p,\lg n-3} + y_{p,\lg n-3} = 2(\lg n - 2).$$

We chose the value of the sum to allow a little flexibility in meeting the constraint on the next (earlier) level. Finally we have the 16 min-size constraints of form

$$x_{p,i} \geq \lg n - 3.$$

If all these constraints are met, one of the subcubes will get an additional excess of  $1/16$ th the boundary, since  $1/8$ th of the pairs on the boundary (in this case 1 such pair) are odd and they all send the extra to that subcube. The even pairs create no additional excess.

In general, the constraints for the level- $(i - 1)$  subcubes come from forcing a bad split for level  $i$  across a boundary of size  $b(i)$  and meeting the input constraints for the subcube that receives the excess. They are of the following form:

$$\begin{aligned} x_{p,i} &= k(x_{p,i}) && [b(i)/4 \text{ force constraints}] \\ x_{p,i} + y_{p,i} &= 2k(x_{p,i}) && [7b(i)/8 \text{ sum constraints}] \\ x_{p,i} &\geq i && [2b(i) \text{ min-size constraints}] \end{aligned}$$

The notation  $k(x_{p,i})$  denotes a fixed value (not a variable), but it is different for each constraint.

Before showing how to find constraints for a level- $(i - 2)$  subcube given constraints for both of its level- $(i - 1)$  subcubes, we first look more closely at the relationships between the

variables  $x_{p,i}$  across different levels  $i$ . Figure 4-1(a) illustrates level by level which processors affect the final load in processor 111 of an 8-processor hypercube. In fact, the same tree applies to processor 110. The numbers in the tree nodes represent processor identifiers. The load, after the  $i$ th exchange, of a processor at height  $i$  (where the leaves are height 0) is determined by the load of its two sons prior to the  $i$ th exchange. On exchange  $i$ , a processor  $p$  is affected by its  $i$ th-dimensional neighbor  $q$ . For example, the final load at processor 111 (after the third exchange) is determined by the load in processors 111 and 110 after the second exchange. Figure 4-1(b) focuses on a level- $(i+1)$  variable, representing the load at processor  $p$  prior to the  $(i+1)$ st exchange, and shows the variables that affect it from the previous two levels. We will refer to these variables in the following discussion.

First we handle the  $b(i)/4$  force constraints of the form  $x_{p,i} = k(x_{p,i})$ . Referring to figure 4-1, we see that the value of variable  $x_{p,i}$  is determined by the values of the two variables  $x_{p,i-1}$  and  $y_{p,i-1}$ . That is, the load of processor  $p$  after the  $(i-1)$ st exchange is determined by its load and its  $(i-1)$ st-dimensional neighbor's load before the exchange. If the pair is even, then the load is evenly split and algorithm A is not called. Therefore we can guarantee the value of processor  $p$  after the  $(i-1)$ st exchange by adding the constraint

$$x_{p,i-1} + y_{p,i-1} = 2k(x_{p,i}).$$

Once we have decided which of the level- $(i-1)$  subcubes (0 or 1) to favor, then we can easily meet the  $b(i)/4 = b(i-1)/8$  force constraints by forming the appropriate  $b(i-1)/8$  sum constraints for the level- $(i-1)$  variables.

Now we try to cause a bad split across the boundary of size  $b(i-1) = 2b(i)$ . The sum constraints offer some flexibility and therefore we can meet them while still causing a bad split. Suppose that we have a sum constraint  $C$  of the form  $x_{p,i} + x_{q,i} = 2k(x_{p,i+1})$  from the 1 subcube. This sum constraint forces the variable  $x_{p,i+1}$ , representing the load of processor  $p$  prior to the  $i+1$ st exchange, to be exactly  $k(x_{p,i+1})$ . Therefore, because of the min-size constraints on all variables, we have that  $k(x_{p,i+1}) \geq i+1$ . Let us assume equality since this assumption is the most restrictive. Thus the constraint becomes

$$\begin{aligned} x_{p,i} + x_{q,i} &= 2(i+1) \\ &= 2i+2. \end{aligned}$$

Therefore, bearing in mind the min-size constraints on  $x_{p,i}$  and  $x_{q,i}$ , we have at least three choices of values for the ordered pair  $(x_{p,i}, x_{q,i})$ , namely  $(i, i+2)$ ,  $(i+1, i+1)$ , and  $(i+2, i)$ .

Referring to figure 4-1(b), we see that there are two pairs of level- $(i-1)$  variables that affect the addends:  $(x_{p,i-1}, y_{p,i-1})$  and  $(x_{q,i-1}, y_{q,i-1})$ . We can allow one of these pairs to be odd, thus affecting the split at level  $i-1$ , and then make the other pair even to meet the sum constraint  $C$ . For example, suppose we force  $(x_{p,i-1}, y_{p,i-1}) = (i, i-1)$ , using the force constraints

$$\begin{aligned} x_{p,i-1} &= i \\ y_{p,i-1} &= i-1. \end{aligned}$$

Then the pair is odd and the favored side receives  $i$  tasks. If algorithm A favors side 1, as we hope, then we have variable  $x_{p,i} = i$ . Therefore, we have  $x_{q,i} = i+2$  from the legal ordered pairs listed above, which we assure by adding the constraint

$$x_{q,i-1} + y_{q,i-1} = 2(i+2).$$

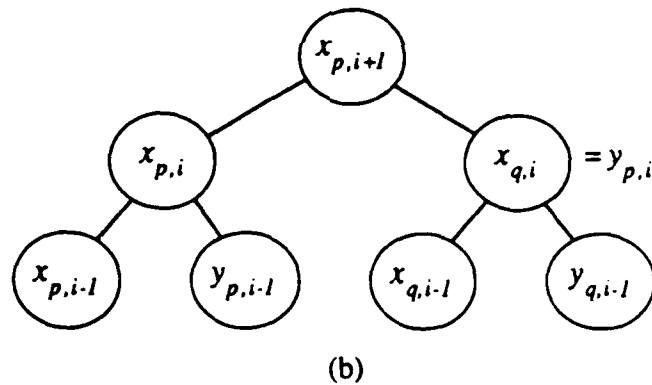
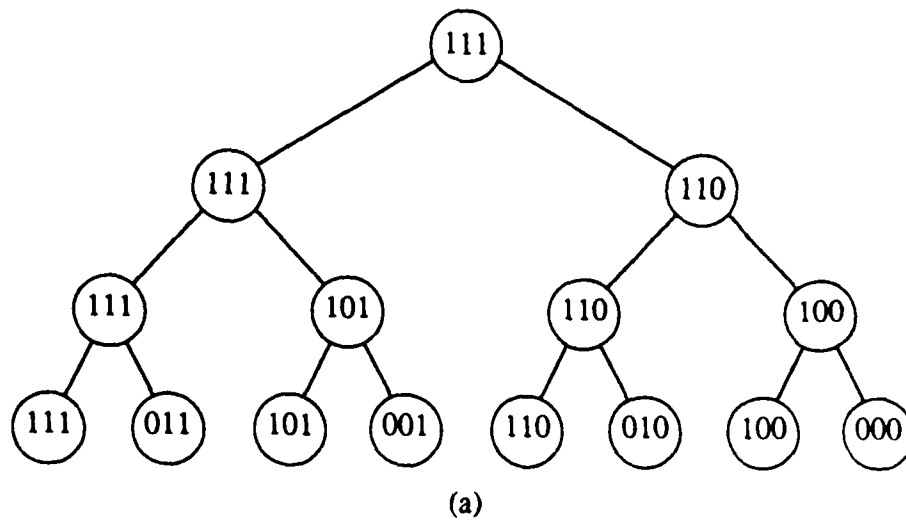


Figure 4-1: (a) By following this tree from the root down to the leaves, we trace the affect each processor has on the final load at processor 111. Nodes are labeled with processor identifiers. Processors at height  $i$  affect their parents on the  $i + 1$ st exchange. (b) We generalize two levels of the tree of (a) to show the relationships between the variables used to set input constraints.

We started the above process by listing all legal values for the ordered pair  $(x_{p,i}, x_{q,i})$ . Then we forced the value of two level- $(i-1)$  variables. We say a pair of such forcing constraints

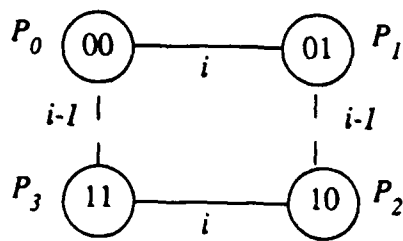
$$\begin{aligned} x_{p,i-1} &= k \\ y_{p,i-1} &= l \end{aligned}$$

is *conditionally sufficient* if  $k+l$  is odd and there exists some  $j$  such that  $(\lceil (k+l)/2 \rceil, j)$  is an ordered pair in our list. An analogous definition applies to forcing constraints on the pair of variables  $(x_{q,i-1}, y_{q,i-1})$ . We say these constraints are only conditionally sufficient because we meet the sum constraint and split in the correct direction only if algorithm *A* makes the choice we want.

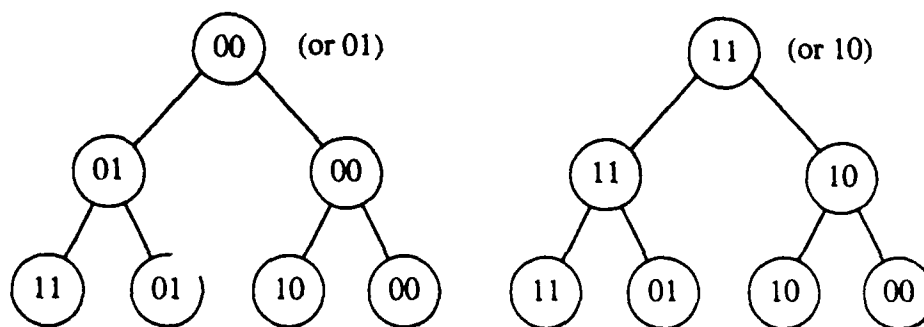
Making sure that we meet the min-size constraints on the level- $(i-1)$  variables, we can list 24 sets of conditionally sufficient forcing constraints. If the min-size constraint on variable  $x_{p,i+1}$  is not tight, then we have even more choices. If algorithm *A* chooses side 1 for any of the 24 processor-input tuples we list, then we can meet constraint *C* while contributing to a bad split at level  $i-1$ . There is always the possibility, however, that algorithm *A* chooses side 0 for all possible choices of conditionally sufficient forcing constraints.

We now argue that we can always meet  $b(i-1)/8$  sum constraints in this manner for at least one of the sides, thus forcing  $b(i-1)/8$  more tasks to that side. First we once again delve into the structure of the dimension-exchange method's communication pattern. Let  $p_0, p_1, p_2$ , and  $p_3$  be four processors whose bits all match except in the  $(i-1)$ st and  $i$ th bits. We call such a set of processors a *quad*. They form a 2-dimensional subcube as illustrated in figure 4-2(a). Pairs  $(p_0, p_2)$  and  $(p_1, p_3)$  communicate on the  $(i-1)$ st exchange, with the first processor of each pair on the 0 side of the communication boundary and the second processor on the 1 side. Then pair  $(p_0, p_1)$  within the side-0 level- $i$  subcube and pair  $(p_2, p_3)$  within the side-1 level- $i$  subcube communicate on the  $i$ th exchange. Figure 4-2(b) illustrates the dependencies among the variables representing the inputs to these processors at levels  $i-1$  and  $i$ . We see that the inputs to *both* pairs that communicate on the  $i$ th exchange — one on the 0 side and one on the 1 side — are determined exclusively by the inputs to the quad at level  $i-1$ . We call such pairs *quad mates*.

We now show that if both quad mates are involved in a sum constraint, then the sum constraint for at least one of the quad mates can be met while forcing an extra task to its side of the communication boundary. Suppose we have a sum constraint  $C_0$  on the inputs to processors  $p_0$  and  $p_1$  before the  $i$ th exchange. We wish to meet this constraint while causing a bad split into the side-0 subcube on the  $(i-1)$ st exchange. According to the method described above, we manipulate the quad  $(p_0, p_1, p_2, p_3)$ , finding all conditionally sufficient forcing constraints. If we have a sum constraint  $C_1$  on the inputs to processors  $p_2$  and  $p_3$  at level  $i$ , then we manipulate the same quad in the same manner. Because the fixed values (right-hand side) of constraints  $C_0$  and  $C_1$  can differ, we may have different conditionally sufficient forcing constraints, depending upon which of constraints  $C_0$  and  $C_1$  we are trying to meet. Looking back at the above description of the method, however, we see that there is always a legal way, relative to all the side-0 constraints, to set the input to processor  $p_0$  to  $i$ , and similarly for processor  $p_2$  on side 1. Thus, the constraints  $x_{p_0,(i-1)} = i$  and  $x_{p_2,(i-1)} = i-1$  are conditionally sufficient for both constraint  $C_0$  and constraint  $C_1$ . Therefore, regardless of the choice algorithm *A* makes, we can meet constraint  $C_j$  for side  $j$  with an extra task going to side  $j$  for either  $j = 0$  or  $j = 1$ .



(a)



(b)

Figure 4-2: (a) A *quad* is a 2-dimensional hypercube consisting of four processors who match in all bits except the  $(i-1)$ st and  $i$ th. These bits are shown within the circles. Pairs communicate along the dashed lines during the  $(i-1)$ st exchange and along the solid lines during the  $i$ th exchange. (b) The dependencies of the variables representing the load at the processors  $P_k$  during the  $(i-1)$ st and  $i$ th exchange. Sons determine the value of their parent. The load of each processor in the quad after the  $i$ th exchange (the root of each tree) is determined by the loads of all processors in the quad prior to the  $(i-1)$ st exchange (the leaves).

We now complete the argument that we can meet  $b(i-1)/8$  sum constraint for one side in such a way that that side receives  $b(i-1)/8$  extra tasks. We first count the number of quads where both quad mates are involved in a sum constraint. In general, we notice that by construction of the constraints for the level- $(i-1)$  subcubes, if a pair of processors  $p$  and  $q$  communicate on exchange  $i$ , then either both variables  $x_{p,i}$  and  $x_{q,i}$  are forced, or they are addends of a sum. Suppose we are given constraints for the two level- $(i-1)$  subcubes, (i.e., constraints on the values of processors in two communicating subcubes before the  $i$ th exchange). Each side has  $7b(i)/8 = 7b(i-1)/16$  sum constraints and  $b(i)/4 = b(i-1)/8$  force constraints. In the worst case, each pair of forces is a quad mate to a sum. In this case there are still

$$\frac{7b(i-1)}{16} - \frac{b(i-1)}{8} = \frac{5b(i-1)}{16}$$

sets of quad mates which are both sum constraints. For each of these sets, we can satisfy the sum constraints for one side while forcing an extra task to that side. Of the  $5b(i-1)/16$  such sets, at least half, or

$$\frac{5b(i-1)}{32} > \frac{b(i-1)}{8}$$

satisfy the same side, which completes the proof of the claim.

Now we describe how to form constraints for a level- $(i-2)$  subcube given constraints for its two level- $(i-1)$  subcubes. We begin by looking at all sets of quad mates which are both sum constraints. We form constraints for  $b(i-1)/8$  of the quads such that the constraints for one side (side 1 without loss of generality) are met and an extra task is sent to that side. This is always possible as proved above. This process gives us  $b(i-1)/4$  force constraints (two for each quad) and  $b(i-1)/8$  sum constraints (one for each quad). Now we proceed to meet the rest of the constraints for side 1, using even pairs so the imbalance is maintained. We meet each of the  $b(i)/4 = b(i-1)/8$  force constraints for side 1 using one sum constraint as described in the beginning of this discussion. Finally, the  $7b(i-1)/16 - b(i-1)/8 = 5b(i-1)/16$  sum constraints that are not used to force a bad split can be met with two new sum constraints. More specifically a sum constraint of the form  $x_{p,i} + x_{q,i} = 2k(x_{p,i+1})$  is met with the constraints

$$\begin{aligned} x_{p,i-1} + y_{p,i-1} &= 2k(x_{p,i+1}) \\ x_{q,i-1} + y_{q,i-1} &= 2k(x_{p,i+1}) \end{aligned}$$

Thus we have a total of  $b(i-1)/4$  force constraints and  $b(i-1)[1/8 + 1/8 + 10/16] = 7b(i-1)/8$  sum constraints, completing the induction on the number of each type of constraint. We also have ensured that side 1 gets an additional excess of  $b(i-1)/8$  on the  $(i-1)$ st exchange, and we have met all the input constraints to that side. Therefore, the new constraints guarantee a bad split from level  $(i-1)$  to the end as we intended.

We continue forming constraints for earlier and earlier levels until finally we have one set of constraints for variables of the form  $x_{p,1}$ . These are constraints on the initial distribution of tasks. If we meet these constraints, then there is a bad split on every level down to a constant-size subcube. The  $n/16$  force constraints are easy. We simply set the initial number of tasks for these processors to the required value. Now we consider one of the sum constraints  $x_{p,1} + x_{q,1} = 2k$ . We can simply use the same arguments we used earlier. The constraint forces  $x_{p,2} = k$  and therefore, by the min-size constraint on the level-2 variable, we have  $k \geq 2$ .

Therefore we can set  $x_{p_1} = 1$  and  $x_{q,1} = k - 1$ . Therefore we can find an initial distribution that forces bad splits on each level.

Using the same arguments as those used in section 4.0.1, we see that the worst-case excess for any level- $k$  subcube after the  $k$ th exchange is given by the following recurrence:

$$\begin{aligned} E(k) &= \frac{E(k-1)}{2} + \frac{n}{2^{k+4}} \\ E(0) &= 0. \end{aligned}$$

This recurrence is valid up to  $k = \lg n - 3$ . Solving the recurrence, we have  $E(k) = kn/2^{k+4}$ , and substituting  $k = \lg n - 3$  yields  $E(\lg n - 3) = \lg n/2 - 3/2$ . This is the excess in the worst-case subcube of size 8. If the load is distributed perfectly from this point on, then each processor in the subcube ends up with excess  $\lg n/16 - 3/16$ . Thus for any algorithm  $A$  we can find a distribution of tasks to processors such that dimension-exchange load balancing with algorithm  $A$  as arbitrator causes at least one processor to have  $\Omega(\lg n)$  excess.

We comment that the initial distribution we construct assigns only a bounded number of tasks to each processor. In practice real machines have only a bounded amount of memory per processor, and therefore each processor can only simulate a bounded number of other processors. Thus our lower bound is achieved with a realistic task distribution.

### 4.0.3 Message Complexity

In this section, we discuss the number of data messages sent during the dimension-exchange load balancing method. We show that the message complexity can be quite high compared to the reduction in the worst-case load.

In a data-parallel architecture, the amount of time required to complete an operation is limited by the maximum number of computations performed by any processor. Thus we would like to reduce this *global maximum load*, and as we saw in section 4.0.1, the dimension-exchange load balancing algorithm does this as well as possible to within an additive  $\lg n$  factor.

Load reduction is not the whole story, however. When we execute the load balancing algorithm on a data parallel architecture, the number of data messages on each exchange is equal to the maximum number of messages sent between any two communicating processors. Thus the performance of the load-balancing algorithm on a given distribution  $D$  is more appropriately measured by the *message rate* which is defined as follows. Let  $M(D)$  be the number of data messages sent during the execution of the load-balancing algorithm on distribution  $D$ , and let  $R(D)$  be the reduction in the global maximum load. Then the message rate is  $M(D)/R(D)$ . It corresponds to the average number of messages sent to achieve a reduction of 1 in the global maximum load. For example, as we saw in section 3.3.1, the implementation of the extract primitive (dimension-exchange load balancing applied to a special distribution) has the optimal message rate of 1. If an initial distribution has message rate  $k$ , then load balancing is effective for that distribution only if a given machine can send a piece of data at least  $k$  times faster than it can compute in place.

We now describe a load distribution that has a message rate of  $\Omega(\lg n)$  on an  $n$ -processor hypercube. Let  $b_1, b_2, \dots, b_{\lg n}$  be the bit representation of a processor's identifier and  $w$  be any even integer. If  $b_i$  is the *last* 1 in the representation, then assign the processor  $w$  tasks if bit  $b_{i-1} = 1$  and 0 tasks if bit  $b_{i-1} = 0$ . For example, if the bit representation is 101100, then the

last 1 is in the 4th position. The preceding bit  $b_3$  is equal to 1 and therefore the processor is assigned  $w$  tasks. If we have  $b_j = 0$  for all  $j > 1$ , then technically bit  $b_{i-1}$  doesn't exist. We assign  $w/2$  tasks to the two such processors.

We start with a distribution for which no processor has more than  $w$  tasks. An exchange between two processors, each holding no more than  $w$  tasks, cannot leave either processor with more than  $w$  tasks. Therefore, throughout the execution of the load balancing algorithm, the worst-case message count on any exchange is  $w/2$  messages, caused by communication between a processor with  $w$  tasks and a processor with 0 tasks.

We now argue that when the load balancing algorithm is executed on the above distribution, each of the first  $\lg n - 1$  exchanges requires the worst-case  $w/2$  messages. It suffices to show that before the  $i$ th exchange, the processor  $P_i$  with identifier

$$\underbrace{00 \dots 0}_{i-1} 11 \underbrace{00 \dots 0}_{\lg n - i - 1}$$

has exactly  $w$  tasks and the processor  $Q_i$  with identifier

$$\underbrace{00 \dots 0}_i 1 \underbrace{00 \dots 0}_{\lg n - i - 1}$$

is empty. These two processors are  $i$ th-dimensional neighbors, so on the  $i$ th exchange processor  $P_i$  sends exactly  $w/2$  tasks to the empty processor  $Q_i$ .

We now argue that processor  $P_i$  has exactly  $w$  tasks just before the  $i$ th exchange. First we look at which processors directly or indirectly communicate with processor  $P_i$  during the first  $i - 1$  exchanges. The load of a processor after the  $i$ th exchange is affected by the initial load of each processor that matches its address in the last  $\lg n - i$  bits. These processors form a subcube of size  $2^i$ . For example, in figure 4-1(a), after the first exchange, the load of processor 111 is the average of the initial loads of processors 111 and 011. After the second exchange, it is the average (ignoring integrality constraints for the moment) of the initial loads in processors 111, 011, 001, and 001. Looking at the figure, these are the leaves of the tree rooted at processor 111 at height 2.

After the first  $i - 1$  exchanges, processor  $P_i$  contains the "average" of the initial load in all processors that match it in the last  $\lg n - i + 1$  bits, namely two 1's followed by  $(\lg n - i - 1)$  0's. Therefore, by the rule described above to assign initial load to processors, each of these processors is initially assigned  $w$  tasks. Since the first  $i - 1$  exchanges pair processors from within this subcube, after these exchanges no tasks have been moved and each processor, including  $P_i$ , still has  $w$  tasks. An analogous argument shows that each processor whose identifier matches that of processor  $Q_i$  in the last  $\lg n - i + 1$  bits is initially empty, and therefore processor  $Q_i$  is empty after the first  $i - 1$  exchanges.

We have just shown that each of the first  $\lg n - 1$  exchanges requires  $w/2$  messages. Therefore we have the total message complexity  $M \geq w(\lg n - 1)/2$ .

We now argue that  $R$ , the global reduction in maximum load, is at most  $w/2$ . We show that the processor  $Z$ , whose identifier is all 0's, has a load of  $w/2$  after load balancing, which is  $w/2$  less than the original maximum of  $w$ . Since its identifier has no 1's, processor  $Z$  is one of the two special processors whose initial load is  $w/2$ . We now show that its load remains unchanged throughout the execution of the load balancing algorithm. On exchange  $i$ , processor

$Z$  communicates with processor  $Q_{i-1}$  having identifier

$$\underbrace{00 \dots 0}_{i-1} \underbrace{100 \dots 0}_{\lg n - i}.$$

Referring to the preceding discussion, we see that processor  $Q_{i-1}$  initially has load 0 and on exchange  $i-1$  the load is increased to  $w/2$ . Thus before its communication with processor  $Z$  on the  $i$ th exchange, both processors have load  $w/2$ , and therefore the  $i$ th exchange does not affect the load of processor  $Z$ .

Therefore we have shown that the load balancing algorithm running on this initial distribution has global reduction  $R \leq w/2$ . Therefore we have a message rate of

$$\begin{aligned} M/D &\geq w(\lg n - 1)/2 \div w/2 \\ &= \lg n - 1, \end{aligned}$$

which is  $\Omega(\lg n)$ . For this distribution, therefore, it would seem the load balancing procedure will not be beneficial unless there is a substantial amount of computation to be performed on each piece of data.

We comment that the message rate as discussed above is for *data* messages. The load balancing algorithm always requires exactly  $\lg n$  *bookkeeping* messages, one for each exchange, so that each processor can determine how many tasks to send.

## 4.1 Conclusions

In this section we offer some concluding remarks including directions for future research.

We showed in this chapter that the dimension-exchange load balancing method can give predictably good balancing. The message complexity, however, is harder to describe completely. For example, what is the average message rate for a random initial distribution? Is there a better set of distributions over which to take this average? Another question is if a similar strategy on another architecture such as the butterfly can remove the  $\lg n$  excess.

The lower bound proof of section 4.0.2 had some restrictions. Of course we would like to remove some of these restrictions. For example, we would like the bounds to apply to dimension-exchange strategies which do not necessarily balance evenly on each exchange. The techniques we used, however, do not easily extend to this case. Intuitively, allowing imbalance on exchanges should not improve the worst case, but formalizing this intuition is difficult. We also did not formalize our intuition that ignoring information about previous exchanges is not a serious restriction.

## Chapter 5

# The Assignment Problem

### 5.1 Introduction

Network optimization problems appear in several areas of application including operations research, transportation, engineering design, financial planning and defense. Such problems are characterized, quite often, by their very large size. Massively parallel computers like the Connection Machine (CM) appear to be well suited for both sparse and dense implementations of dual relaxation algorithms for network optimization. In this chapter we summarize recent experiences with the solution of large scale network optimization problems using the CM. In particular, we discuss key features of a parallel implementation of Bertsekas' algorithm for *assignment* and present results with numerical experiments. Portions of this chapter represent joint work with Stavros Zenios of the Wharton School at the University of Pennsylvania.

A network optimization problem is defined as follows:

$$\begin{array}{ll}\text{Minimize} & F(x) \\ \text{subject to} & A \cdot x = 0 \\ & l \leq x \leq u\end{array}$$

The constraint matrix  $A$  has the special structure of two non-zero entries in every column: a  $+1$  and a  $-1$ . The graph underlying this optimization model can be derived by associating rows of the constraint matrix with a set of nodes  $V = \{1, 2, 3, \dots, n\}$  and columns of  $A$  with a set of directed edges  $\mathcal{E} = \{(i, j) \mid i, j \in V, \text{ and some column of } A \text{ has a } +1 \text{ in row } i \text{ and a } -1 \text{ in row } j\}$ . The model can be written in algebraic form as:

$$\begin{array}{ll}\text{Maximize} & \sum_{(i,j) \in \mathcal{E}} f(x_{ij}) \\ \text{subject to} & \sum_{(i,j) \in \mathcal{E}} x_{ij} - \sum_{(k,i) \in \mathcal{E}} x_{ki} = 0 \\ & l_{ij} \leq x_{ij} \leq u_{ij}\end{array}$$

There are several standard ways to model the real world using the graph underlying this mathematical formulation. We can think of the graph as a plumbing network with the nodes representing joints and the edges representing pipes. Alternatively, we can think of the network as a system of roads (edges) and junctions (nodes). The variables  $x_{ij}$  represent a quantity called *flow*, such as the flow of water through pipes or the flow of products over a road system. The first set of constraints represents conservation of the flows  $x_{ij}$  at all nodes  $i \in V$ . That is, over time the amount of flow into a node is equal to the amount of flow leaving that node. The second set imposes upper and lower bounds,  $u_{ij}$  and  $l_{ij}$  respectively, on the flow over each edge. The upper bounds physically model the *capacity* of an edge, such as the width of a pipe or the weight limit of a road.

There are several well-studied special cases of network optimization. In the *maximum (max) flow problem*, there are two special nodes, a *source* and a *sink* from which the conservation constraints are removed. The source can produce excess flow and the sink can absorb excess flow. We wish to maximize the amount of flow traveling from the source to the sink, subject to the capacity and conservation constraints in the rest of the network. Thus for this case we have

$$f(x_{ij}) = \begin{cases} 1 & \text{if } i \text{ is the source} \\ 0 & \text{otherwise} \end{cases}$$

In the *min cost flow problem*, we associate with each edge  $(i, j)$  a constant cost per unit flow  $c_{ij}$ . We wish to find a maximum flow of minimum cost. Thus we have  $f(x_{ij}) = -c_{ij}x_{ij}$ . We use the lower bounds  $l_{ij}$  and negative costs to prevent optimality for the trivial zero flow where  $x_{ij} = 0$  for all  $i$  and  $j$ .

In the *assignment problem*, also known as *maximum weight bipartite matching*, the set of vertices of the graph  $V$  is defined as  $V = P \cup O$ , where  $P = \{p_1, p_2, p_3, \dots, p_n\}$  and  $O = \{o_1, o_2, o_3, \dots, o_n\}$ . Each edge  $(p, o)$  in the network connects a  $p \in P$  to a node in  $o \in O$ . A typical modeling use of this problem is the one-to-one matching of persons from the set  $P$  to objects or tasks in the set  $O$ . Later in this chapter we will be referring to the sets  $P$  and  $O$  as people and objects respectively. Each edge  $(i, j)$  has a weight  $v_{ij}$  which indicates the value of assigning person  $i$  to object  $j$ . The optimization problem is to choose the assignment that satisfies  $\text{Maximize}_{x_{ij}} \sum_{(i,j) \in E} v_{ij}x_{ij}$ . The variable  $x_{ij}$  has the value 1 if person  $i$  is assigned to object  $j$  and zero otherwise. The assignment problem can be reduced to max flow by giving each edge  $(i, j)$  a capacity  $u_{ij} = 1$ , adding a source node  $s$  connected to each node  $p \in P$  via an edge of capacity 1, and adding a sink node  $t$  connected to each node  $o \in O$  via an edge of capacity 1. Because the assignment problem is a special case of max flow, however, we can use an algorithm adapted for the special constraints of the problem.

Researchers have been able to design and implement efficient algorithms for the solution of extremely large network problems by capitalizing on the special structure of the network basis. General references on network optimization are Kennington and Helgason [70] and Dembo, Mulvey and Zenios [34].

The efficiency with which network problems can be solved prompted the modeling of large and complex systems, such as traffic assignment, air-traffic control, real-time defense systems or cash-flow management, that were considered intractable with general purpose optimizers. Several of these models are envisioned to extend to millions of variables and large time horizons, but unfortunately can not be solved even with network specialized algorithms on large mainframes. Researchers have been turning to parallel and vector supercomputers as a pos-

sible way to solve even larger instances of network models. For general references on parallel optimization, including papers on network problems, see Meyer and Zenios [87] and Zenios [129].

In this chapter we report on the implementation of network optimization algorithms on the Connection Machine (CM-2) hypercube multiprocessor. We now summarize recent experiences with flow algorithms and then in the remainder of the chapter we discuss in detail an implementation of Bertsekas' algorithm for the assignment problem.

Goldberg's algorithms for max flow and min cost flow are well-suited for the Connection Machine architecture because the inner loop can be computed with segmented scans and collision-free communication. Goldberg [49] reports encouraging preliminary experience with his max flow implementation on the Connection Machine. An implementation of his min-cost flow algorithm on the Connection Machine has not been tested or tuned enough to allow commentary other than to say that it suffers from a long sequential tail at the end of each major iteration. Small quantities of flow are sequentially pushed around the network until they finally settle where they belong. We are investigating methods for cutting the tail.

Zenios and Lasken [128] have investigated *strictly convex network problems*. This subclass of network problem is the most general that we have discussed, since now the objective function  $f$  need not be constant, but can be any strictly convex function of  $x_{ij}$ . They implemented a relaxation algorithm for these problems proposed by Bertsekas, Hossein and Tseng[12]. Bertsekas and El Baz [10] and Zenios and Mulvey [130] showed that this algorithm is well suited for massively parallel computations.

Zenios and Lasken use the sparse network representation described in section 5-1. Each inner loop requires segmented scans, local computation, and one collision-free communication. This algorithm was evaluated empirically on both a CM-1 and a CM-2 using two sets of test problems. One set came from a materials-science application. The other set consisted of large randomly generated transportation problems. The same test problems were solved using an inherently serial algorithm, the primal truncated Newton of Ahlfeld et al.[4], on a variety of machines including an IBM 3081-D large mainframe, an IBM 3090-600/VF supercomputer, and an Alliant FX/8 shared memory vector multiprocessor. On the last two machines significant effort went into modifying the software to take advantage of both the vector and parallel features of the hardware. The Connection Machine implementation was competitive with the best alternative implementation for moderate size problems and it was clearly superior for large problems. For example, one of the transportation problems with 2500 nodes and 8000 edges required 1.5 minutes on a vector supercomputer, but less than a second on the Connection Machine.

In the remainder of this chapter, we discuss the solution of assignment problems on the Connection Machine. We implemented the *auction algorithm* for solving the assignment problem due to Bertsekas [11]. Though a parallel implementation is not provably asymptotically superior to a sequential one, the algorithm does exhibit heuristic parallel speedup. Section 5.2 discusses sparse and dense representations of network problems. Section 5.3 describes the Bertsekas algorithm and its implementation on the Connection Machine. Section 5.4 describes the "sequential tail" phenomenon and some heuristic measures we used to partially overcome it. Section 5.5 presents the results of our preliminary testing of this implementation. Section 5.6 offers some concluding remarks.

## 5.2 Representing Networks on the Connection Machine

In this section we introduce the dense representation of the assignment problems and an alternative sparse implementation used, for example, in the Zenios/Lasken implementation of nonlinear network optimization. The features of the CM which are relevant to the following discussion are described in section 1.3. For further details of the architecture of the CM, see [57] and system documentation.

### 5.2.1 Representation of Dense Assignment Problems

Other than the simplex linear programming implementation discussed in chapter 3, all CM network optimization implementations to date have been represented sparsely. Considering the size of the problems we wish to run, this is a wise choice of representation. For the assignment problem, however, we use a dense representation, in part because we wish to exercise the architectural features of the Connection Machine on dense problems, and in part because we have a dense problem from a real application upon which to test our implementation.

Because we assume that the input graph is dense, we configured the CM as an  $n \times n$  NEWS grid, where  $n$  is the number of people or, equivalently, the number of objects, rounded up to the closest power of 2. Row  $i$  is associated with person  $i$  and column  $j$  is associated with object  $j$ . In particular, processor  $(i, j)$  stores the value  $v_{ij}$  of object  $j$  to person  $i$ , local variables applicable to person  $i$ , and local variables applicable to object  $j$ . The variables used in the implementation are described in section 5.3.

### 5.2.2 Representation of Sparse Network Problems

The representation of sparse network problems chosen here is motivated by the operations typically used in relaxation algorithms for network optimization problems. Processors associated with a single node are grouped, thus we can use scans to communicate within nodes, bypassing the router. Communication across edges still requires the router, but it is guaranteed to be collision free. This data structure was introduced by Blelloch [15] for use in general algorithms for sparse graphs.

Figure 5-1 shows the representation of a simple undirected network. The array in the figure represents the processors of the CM in linear order. Every edge in the network is associated with two processors, one for each endpoint, and the processor for each endpoint holds a pointer to the processor for the other endpoint. To allow use of segmented scan operations, processors associated with node  $i$  are grouped into contiguous segments. These segments are separated by heavy lines in the figure. We can represent directed networks in the same manner except that the pointers need not be bidirectional. If an edge exists from node  $i$  to node  $j$  but not in the other direction, then one processor in the segment for node  $i$  holds a pointer into the segment for node  $j$ , but there is no return pointer.

Using this representation, all nodes can communicate simultaneously to all adjacent edges using segmented scans and collision-free routing. For example, for each node to determine the maximum identifier of any neighbor, each processor sends its identifier across the pointer it holds, and then a segmented max scan within the node segments simultaneously determines the max for each node.

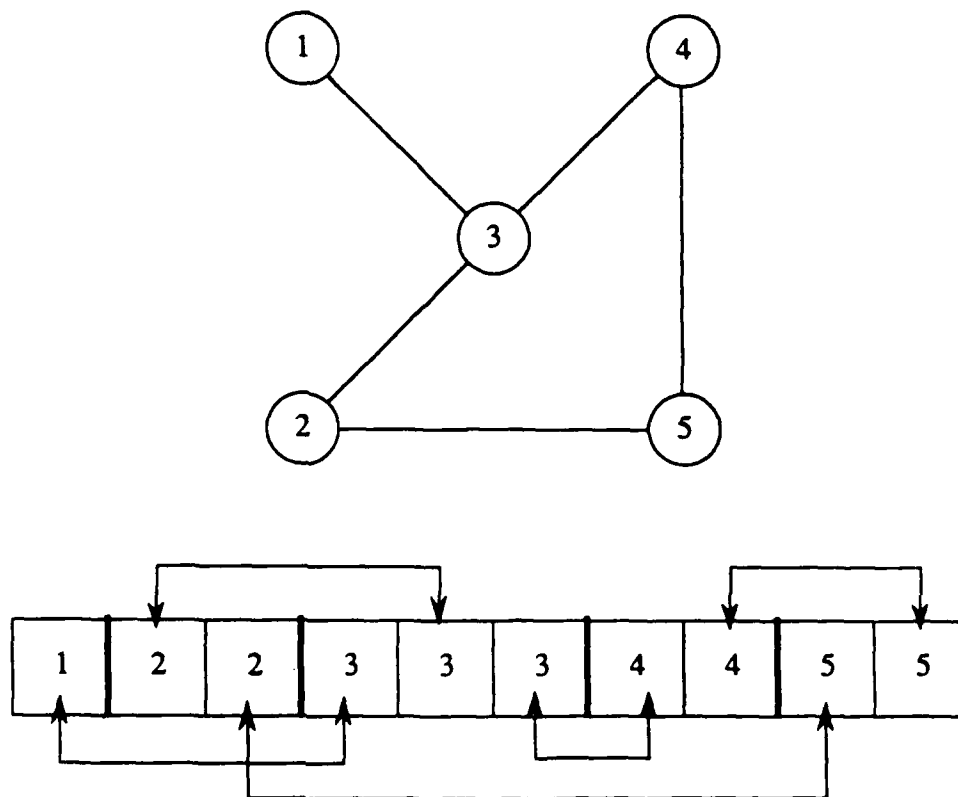


Figure 5-1: Representation of Sparse Network Problems. The array represents the CM processors in linear order. Processors associated with endpoints of an edge point to each other. The heavy lines represent segment boundaries.

Had we implemented the assignment algorithm using this representation, each edge would have one processor  $p_1$  associated with a person  $p \in P$  and the other processor  $p_2$  associated with an object  $o \in O$ . Both processors  $p_1$  and  $p_2$  would hold variables associated with the edge  $(p, o)$ . Processor  $p_1$  would also hold variables associated with person  $p$  and processor  $p_2$  would hold variables associated with object  $o$ . We would use segmented linear scans in place of grid scans and we would use routes whenever the columns of the grid communicate with rows and vice versa.

### 5.3 Bertsekas' Algorithm on the Connection Machine

This section gives a high-level overview of Bertsekas' algorithm for the assignment problem and our implementation of the algorithm on the Connection Machine CM-2. Bertsekas [11] proves the correctness of the algorithm.

Referring to the definition of the assignment problem, we see that in a globally optimal solution, any given person may not be assigned to the object which is most valuable to him. For any optimal assignment, however, it is possible to assign to each object  $j$  a price  $\pi_j$  such that if each person  $i$  views the profit of being assigned to object  $j$  as  $v_{ij} - \pi_j$ , then every person is assigned to the object that is most profitable. The prices  $\pi$  serve as dual variables in the linear programming sense. An assignment is  $\epsilon$ -optimal if each person is assigned to an object that is no more than  $\epsilon$  less profitable than its most profitable object. That is, the person is not totally happy, but he is only  $\epsilon$  off his best.

The auction algorithm is performed in waves. We start with  $\epsilon = \max_{i,j} v_{ij}$  and find an  $\epsilon$ -optimal assignment. This initial value for the parameter  $\epsilon$  is large enough to guarantee that any assignment is  $\epsilon$ -optimal. Keeping the prices  $\pi$  generated in the first round, we then halve  $\epsilon$  and find a new assignment optimal for that  $\epsilon$ , and so on until  $\epsilon < 1/n$ , where  $n$  is the number of people. At this point, an  $\epsilon$ -optimal assignment is globally optimal [11]. In practice, we scale all values by  $n + 1$  at the start to allow integer calculations throughout.

Each inner loop (finding an  $\epsilon$ -optimal assignment) can be viewed as an auction where all currently unassigned people bid on their most profitable object, the price of each object is raised to the highest bid, and each object bid upon is temporarily assigned to the highest bidder.

More precisely, the auction algorithm is implemented as follows:

#### The Assignment Algorithm

**Step 0: Initialize.** In particular set up the value of  $\epsilon$ , the error parameter, and scale where needed.

**Step 1: Determine if everyone is assigned.** If a person is unassigned, proceed to step 2. If every person is assigned to an object and  $\epsilon \leq 1$ , the algorithm terminates. Otherwise, if  $\epsilon > 1$ , reduce its value and unassign everyone whose current assignment is no longer  $\epsilon$ -optimal for the new  $\epsilon$ .

**Step 2:** Select only processors associated with unassigned people. Using grid scans, within each row find the column index  $b$  of the best (most profitable) object and the profit  $p$  of the next best object. Each unassigned person  $i$  now bids on his best object  $b$  by setting a variable in processor  $(i, b)$  to  $[v_{ib} - p + \epsilon]$ . The bid function has the property that if

the price of the best object is raised to the bid and person  $i$  is assigned that object, the assignment will be  $\epsilon$ -optimal at the time of the assignment. We can verify the property by computing the profit to person  $i$  of his best object once its price is raised to the bid:

$$\begin{aligned}\text{profit}(i, b) &= v_{ib} - \pi_b \\ &= v_{i,b} - v_{i,b} + p - \epsilon \\ &= p - \epsilon.\end{aligned}$$

If the object with the second best profit  $p$  is also bid upon during this phase, then its profit margin will drop and this person is even closer than  $\epsilon$  to his best profit margin.

**Step 3:** Using a max grid-scan within all columns, determine the maximum price bid on each object and update the prices  $\pi$  within the columns. For all objects bid upon, undo any previous assignment and assign the object to the highest bidder. Go to step 1.

A few comments about the implementation described above are in order. It is possible to perform a scan within an axis of a grid and tell each processor within that axis the result obtained by the last processor. At first it seems one must do a forward scan followed by a reverse copy scan, but the behavior can be provided by a single scan that requires no more time than an ordinary scan. This is the behavior obtained by using the *reduce* and *distribute* primitives described in chapter 3. Secondly, in determining the winning bid, we concatenate the row number to the bid during the scan to break ties. Furthermore, with this concatenation we know both the winning bid and the row number (given by the least significant digits of the concatenated number). Using similar concatenation, we can find the column number of the most profitable object for each row using a single scan.

## 5.4 Tails and Tail Cutting

In this section we discuss the parallelism of Bertsekas' algorithm. We demonstrate the existence of a sequential *tail* at the end of each auction round and we discuss heuristic measures we employed to cut this tail.

The inner loop of each auction round appears highly parallelizable in that all people vying for objects can compete simultaneously. The Connection Machine performs the inner loop in a time that is independent of the number of people actively seeking objects. That is, the CM can update the matrix of values for 1000 active people in the same amount of time it can update for 2 active people. In terms of work accomplished in unit time, this is a wonderful property if many people are active most of the time. Because the Connection Machine processors are not very powerful, however, this property is not so wonderful if few people are active.

At the start of each auction with a new  $\epsilon$ , many people bid simultaneously, but the number of people bidding decreases monotonically during the auction. Near the end of each wave of the algorithm (each auction), there is frequently a rather long sequential tail where only a very small number of people are bidding. For example, we ran a  $1000 \times 1000$  problem from a military application. Although up to 1000 people can actively bid on each iteration, for almost 39% of the iterations only one person bids. Furthermore, for 82% of the iterations at most 10 bid and for 96% of the iterations at most 100 bid. Thus for virtually the entire run we use only 10% of the CM's capacity, and, in fact, for a vast majority of the time we use only 1%.

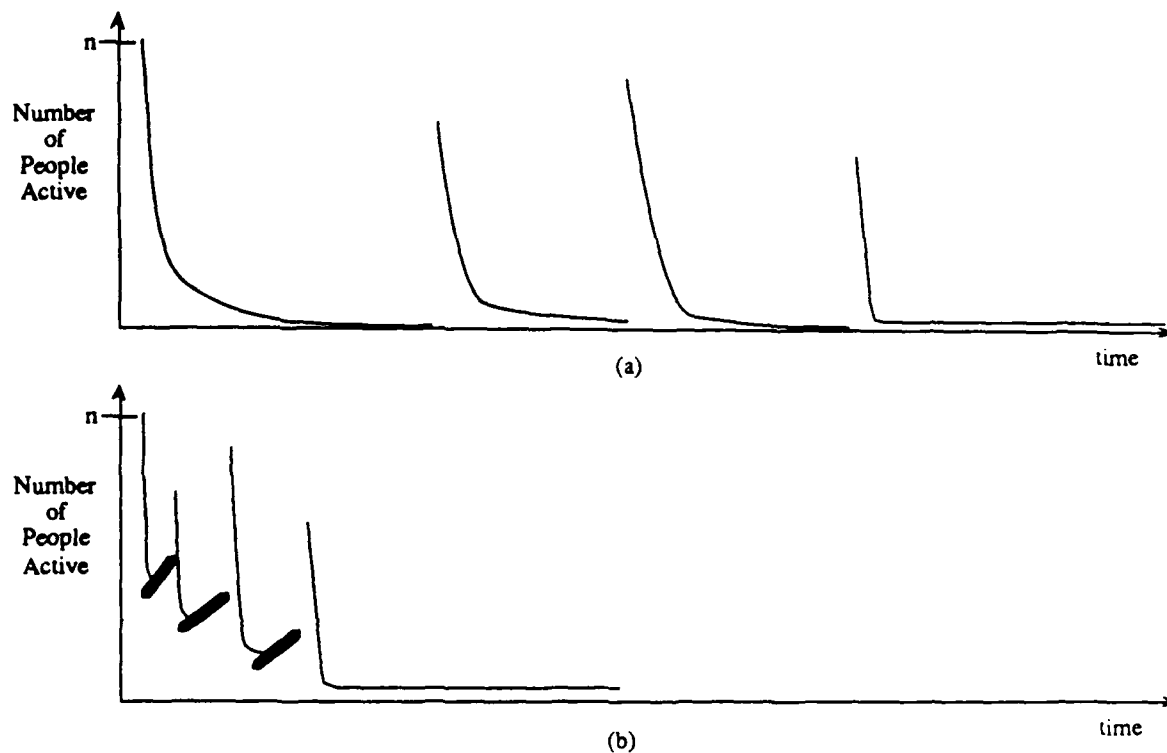


Figure 5-2: (a) A schematic view of the activity pattern of a typical execution of Bertsekas' algorithm. During each auction (the search for an  $\epsilon$ -optimal assignment), the number of active people decreases monotonically, and each auction ends with a long "tail" where very few people are active. (b) We improve the running time of the algorithm considerably by cutting the tail in all but the last auction. The user specifies a parameter  $k \leq 1$  that tells how much to cut the first tail. We stop when  $kn$  of the  $n$  people are matched. We then let each successive auction run a little longer, until the last which is run to completion.

Because on the first iteration, at least, all 1000 people bid, we conjecture that the activity pattern of a typical run consists of a series of tails as shown schematically in figure 5-2a. Each curve represents the number of people active over time during an auction for some  $\epsilon$ . During each auction the number active decreases monotonically until at the end only a few active people — ultimately only one — remain, and the auction reduces to sequential "bumping". That is, the one active person bids an object away from another person, who in turn bids another object away from someone, and so on until finally the last remaining person bids upon the one unassigned object. After each auction, the error parameter  $\epsilon$  is reduced and all people whose assignments are no longer  $\epsilon$ -optimal are unassigned, and therefore the number of active people jumps upward (in an unpredictable fashion).

We use two heuristics for improving performance. First of all, for the special case of only one person bidding, we can improve the CM performance by substituting global operations for grid scans and computing on the front end. Secondly, we allow incomplete matchings for early auctions. More precisely, we allow the user to specify an initial value for a *tail-cutting parameter*

$k \leq 1$ . For a given value of  $\epsilon$ , the auction round terminates when  $kn$  of the people have been matched. The parameter  $k$  increases as  $\epsilon$  decreases until  $k = 1$  when  $\epsilon < 1$ . That is, the final auction is run to completion. If we set  $k = 1$  initially, we are back to the original algorithm. To assist in fine-tuning the algorithm, we allow the user to specify the amount **dec-factor** by which the parameter  $\epsilon$  is divided in each iteration, thus determining the number of auction rounds to be performed by the algorithm. The tail-cutting parameter is initially equal to the user-specified parameter  $k$  and is increased uniformly. That is, if there are  $r$  rounds of the algorithm, then on each iteration, the tail-cutting parameter is increased by  $\lceil(1 - k)/r\rceil$ . Figure 5-2b shows the tail-cutting procedure schematically. Because each auction in general starts with an arbitrary number  $m \leq n$  of people matched anyway, this tail-cutting does not affect the correctness of the algorithm.

Cutting the tail improved the performance of the algorithm dramatically. For example, solving the  $1000 \times 1000$  military example mentioned earlier using Bertsekas' algorithm without tail-cutting required about 50 minutes of time on an 8224-processor CM-2. After choosing appropriate values for the tail-cutting parameters, the same problem was solved in less than a minute on the same size machine.

## 5.5 Experimental Results

In this section we describe the results of running the auction algorithm on test problems of varying sizes and value ranges. We found that the algorithm was quite sensitive to the choice of parameters  $k$  and **dec-factor**. We describe the nature of this sensitivity and speculate on choices that seem to work well in general.

In particular, we will give some evidence to back up the following rules of thumb: If you know nothing about a problem, try **dec-factor** = 2,  $k = 0.90$ . For problems with small value ranges, try intermediate values of parameters **dec-factor** and  $k$ , for example values of **dec-factor** in the range 4–7 coupled with  $k = 0.85$  or values of **dec-factor** in the range 8–11 coupled with  $k = 0.85$  or  $0.90$ . For problems with larger value ranges, try low values of **dec-factor** such as 2 or 3 coupled with high values of  $k$  such as 0.9 and 0.95.

We used the auction algorithm to solve the assignment test problems shown in table 5.1. The first problem, ASSIGN1, is the military application mentioned in the previous section. The other problems were created randomly and locally using the Connection Machine random number generator. All problems are complete (dense), meaning that an edge exists between each pair of nodes  $(p, o)$  with  $p \in P$  and  $o \in O$ . We ran all the  $1000 \times 1000$  problems on a 16,448-processor CM-2 running at 6.7 MHz. Problem ASSIGN2, which is  $500 \times 500$ , was run on an 8224-processor CM-2 with the same clock speed.

We remind the reader that the parameter **dec-factor** is the number by which  $\epsilon$  is decremented after each auction, and the parameter  $k$  specifies how many people will be matched in the first auction. In an attempt to determine good general values for the parameters, we tried many values. In general, we ran each problem for the parameters **dec-factor** = 2, 3, ..., 10 and  $k = 0.65, 0.7, \dots, 0.95$ . Some of the problems were run with more values of the parameter **dec-factor**, and problem ASSIGN1 was run only for initial values of the tail-cutting parameter  $k = 0.8, 0.85$ , and  $0.9$ .

Bertsekas' analysis of the algorithm guarantees only a worst-case sequential complexity of  $O(ne \log(nv))$  where  $n$  is the number of nodes,  $e$  is the number of edges, and  $v$  is the maximum

Dense Assignment Problems		
Problem	No. Nodes	Value Range
ASSIGN1	1000 × 1000	1-10
ASSIGN1a	1000 × 1000	0-9
ASSIGN1b	1000 × 1000	0-99
ASSIGN1c	1000 × 1000	0-999
ASSIGN1d	1000 × 1000	0-9999
ASSIGN2	500 × 500	0-999

Table 5.1: We performed the auction algorithm on these complete bipartite graphs. Problem ASSIGN1 was derived from a military application. The rest of the assignment test problems were randomly generated. Value range refers to the values  $v_{ij}$  of the objects.

absolute value of any  $v_{ij}$ . Parallelism offers no asymptotic theoretical improvement, and the most we can gain heuristically is a factor of  $n$ . Thus for unfortunate choices of the parameters, we could have very long running times. We began with the intent of determining the best values of the parameters for a given problem. Given the potential for very bad runs, we stopped those that were clearly inferior to previous runs for the same problem. Our testing strategy evolved to a more automated form whereby a run was stopped after the number of bid-making iterations was twice the number used by the best previous run. By automating the termination procedure, we were able to obtain the value of the assignment at the time of termination, which led us to ask the following question.

What if we cut the tail in the final auction as well as the earlier ones, and then arbitrarily assign the remaining people to the remaining objects? The latter step is easy to implement on the Connection Machine. We simply use the scanning facility to enumerate the remaining people and objects and then assign the  $i$ th remaining person to the  $i$ th remaining object. If an application requires merely a nearly optimal solution, then this procedure can be expected to save a considerable amount of time, provided that the assignments generated early in the auction algorithm are good.

We did not investigate this strategy, but the data we gathered incidental to our investigation of fully optimal assignments offers some insight. For the runs that were terminated early, values of the partial assignments are consistently good, which is especially encouraging given that

1. we did not arbitrarily complete the assignments, and therefore some unmatched people contribute nothing to the value of the assignment.
2. we looked only at "bad" values of the parameters, because otherwise the run would not have been terminated, and
3. the runs could be stopped right at the beginning of a new auction when many people have just been unassigned.

We first look at the dependence of the algorithm on the degree to which we cut the tail, reflected in the choice of parameter  $k$ . We then look at the dependence upon the number of

Inner Loop Timings	
VP Ratio	CM time
16	.0232
32	.0459
64	.0855

Table 5.2: Timings of the scan-based version of the inner loop of the auction algorithm taken at various virtual-processor ratios. We drew the best line through these points and used it to estimate the speed-up we can anticipate with a full machine.

auction rounds, reflected in the choice of parameter **dec-factor**. For the runs that produced optimal assignments, we look at the time required for solution. For the runs that were terminated early, we look instead at the value of the assignment at the time of termination. Not surprisingly, they showed the same general dependence.

To allow the comparison of timings made on different size problems on different size machines, we normalize all the times for a full 65,792-processor machine. Scaling does not effect the comparison of runs on the same problem since all such times are changed equally. The key to the comparison of algorithms run on different size CM's is the *virtual-processor (VP) ratio*, which is the number of processors simulated by each physical processor. Any CM requires about the same amount of time to perform an operation at a given VP ratio regardless of the physical size of the machine. Larger machines take slightly longer to communicate, since signals must propagate longer distances, but we will ignore this effect. The  $1000 \times 1000$  problems were run on a 16,448-processor machine, which requires a VP ratio of 64 : 1. The  $500 \times 500$  problem was run on an 8224-processor machine which requires a VP ratio of 32 : 1. On a full machine the  $1000 \times 1000$  and  $500 \times 500$  problems require VP ratios of 16 : 1 and 8 : 1 respectively.

To arrive at our estimate of times on a full machine, we timed the inner loop of the auction algorithm at three different virtual processor ratios. The results are shown in table 5.2. We then drew the best line through these points and used it to estimate the factor by which the times would decrease on a full machine. In general on the Connection Machine, times scale sublinearly with VP ratio, especially at high ratios. We looked at the behavior in the range of interest and determined that the  $1000 \times 1000$  problems would run about 3.58 times faster on a full machine and the  $500 \times 500$  problems would run about 3.21 times faster. The timings were made on the scan-based version of the inner loop. The version that uses global operations scales more linearly with VP ratio. Because we do not know how many of each kind of iteration occurred on each run, we chose to be conservative and base the scalings on the scan-based version.

In the following discussion, all times are the *CM time*. The actual running time includes the time the CM must wait for instructions and/or computations on the front end computer. As one might expect, the total time is strongly correlated with the CM time. Some variation is caused by the use of the two different implementations of bidding. Bidding iterations that use global operations are much faster than the scan-based iterations in both total time and CM time. The percent of the total time in which the CM is active (*percent utilization*) decreases, however, since the CM must now spend much more time synchronizing with and waiting for

the front end. We chose to use the CM time for comparisons because it is less sensitive to the choice of front end. Also, once we determined that the computation was real-time on the CM, we became primarily concerned with the behavior of the algorithm relative to the parameters, thus comparing the CM to itself rather than to other supercomputers.

In discussing partial-assignment quality, we normalize to percent error from optimal. The percent error was uniformly excellent. For almost all test cases, the error was much less than one percent.

Table 5.3 summarizes the performance of the algorithm across all test problems. There was only one setting of the (**dec-factor**,  $k$ ) pair which ran to completion on all test cases, namely (2,0.9). If we know nothing about a problem, these values are a good starting point since with them the algorithm is likely to terminate in real time. The table reports the CM time for this setting of the parameters.

The table also reports the best time for each problem, along with the parameter settings that realized that value. The timings for problems ASSIGN1a-d, illustrate a property of the algorithm that will be more obvious later in this discussion: the performance of the algorithm is quite sensitive to the value range of the problem. These problems are all random  $1000 \times 1000$  with value value ranges increasing from 1-10 for ASSIGN1a up to 1-10,000 for ASSIGN1d. We were somewhat surprised to find that problem ASSIGN1d (range 1-10,000) had a best run that was better than the best run for problem ASSIGN1c (range 1-1000). We believe, however, that it is not indicative of expected performance, since the other settings of the parameters did not perform anywhere near as well at the best ones for this problem. This dependence on value range is not surprising since the initial value of  $\epsilon$ , and therefore the number of auction rounds, is directly proportional to the maximum  $v_{ij}$ . Also, Bertsekas [11] gives an example of a problem where the number of bidding iterations is directly related to the maximum value. The degradation in performance with increases in the value range is not as dramatic as those observed with sequential implementations [11]. As we show later in this section, if we know the cost range, we may be able to make a wiser choice for the parameters than (2,0.9).

Figure 5-3 illustrates the dependence of running time upon the tail-cutting parameter  $k$ . We show the average times for problem ASSIGN1a ( $1000 \times 1000$ , range 1-10) and ASSIGN2 ( $500 \times 500$ , range 1-1000). The former problem is indicative of the behavior for problems with small value ranges and the latter is indicative of the behavior for larger ranges. The average for each  $k$  is taken over those values of the **dec-factor** parameter for which most runs completed. Taking such an average raises the question of how one deals with the few runs which did not complete for these "good" values of **dec-factor**. If we ignore them, then we penalize for runs that completed just before the termination. Unfortunately, running time for the incomplete problems is not always comparable. Rather than monitoring the CM's use of time (and possibly affecting that time by trying to watch it), we found it easier to count iterations. We stopped a run after it had taken twice as many bidding iterations as the best previous run for that problem. Therefore, the termination criterion evolved as we tested each problem. The number of allowed iterations dropped until the best set of parameters was run. For problems with large cost range, the best run was frequently very early and most runs were cut off after the same number of iterations. For these problems, for the runs that did not complete, we averaged in the time they ran until automatically terminated. Problem ASSIGN1a was tested before our automatic cut-off and therefore inferior runs were stopped by hand. For these runs, we average in 120 seconds, which is just longer than the time required by the worst

Best General Parameter Settings			
Problem	CM time	Best time	
	df= 2, k = 0.90	CM time	df/k
ASSIGN1	30.30	6.97	8/0.85
ASSIGN1a	15.05	4.84	10/0.90
ASSIGN1b	14.78	10.31	2/0.80
ASSIGN1c	62.31	21.30	2/0.95
ASSIGN1d	12.25	11.01	2/0.70
ASSIGN2	8.02	4.08	5/0.85

Table 5.3: An overview of suitable parameters for the solution of our test assignment problems. The first column gives times for a set of parameters for which our program ran to optimality for all test cases. CM time is measured in seconds and normalized to a full (64K) machine. The abbreviation **df** stands for **dec-factor**. Notice that the solution time appears to increase with value range as shown by problems ASSIGN1a-d, all random  $1000 \times 1000$  problems with increasing value ranges.

completed run. The termination criteria for Problem ASSIGN1 changed too frequently to allow reasonable comparison, so we did not compute the averages for this problem. Problems ASSIGN1b, ASSIGN1c, and ASSIGN1d had so few optimal runs that at most one value of **dec-factor** finished for a majority of  $k$  values.

For problem ASSIGN1a, the curve in figure 5-3 truly reflects the typical behavior across all values of **dec-factor**. That is, running times for low values of  $k$  are mediocre for most values of **dec-factor**. Times improve steadily as  $k$  increases until they bottom out at  $k = 0.85$  or  $k = 0.9$ . Then they get rapidly worse. For example, for **dec-factor** = 5, the best time occurs at  $k = 0.85$  and the runs for  $k = 0.9, 0.95$  were stopped. In fact, for all values of **dec-factor** except  $dec = 2$ , the run for  $k = 0.95$  was stopped. The average value for  $k = 0.9$  is unimpressive, because for runs at low and high values of **dec-factor**, it did not seem to perform well. For values of **dec-factor** between 8 and 11, however, it performed extremely well, yielding the four best times overall. Problem ASSIGN1 has the same value range as problem ASSIGN1a, namely 1-10, although it is not random. Even though we had few data points, the "curves" follow the same pattern.

Problem ASSIGN2 shows wide fluctuations in running time for the low values of  $k$ , but it had consistently good times for high values of  $k$ . In fact, in marked contrast to the problems with lower value range, for values of **dec-factor** greater than 4, runs finished only for  $k \geq 0.85$ . There was only one example where some value of  $k$  completed but  $k = 0.95$  did not.

We now attempt to draw some conclusions about the dependence on  $k$  of the problems with larger value ranges. These problems had such a strong dependence upon the **dec-factor** parameter that only runs with very low **dec-factor** completed, and therefore we cannot adequately analyze dependence upon  $k$  for optimal assignments. We do observe, however, that in problem ASSIGN1b, for values of **dec-factor** greater than 4, the only runs that ran to completion had  $k = 0.95$ . Thus it seems that the behavior of problem ASSIGN2 is more typical of problems with larger value ranges.

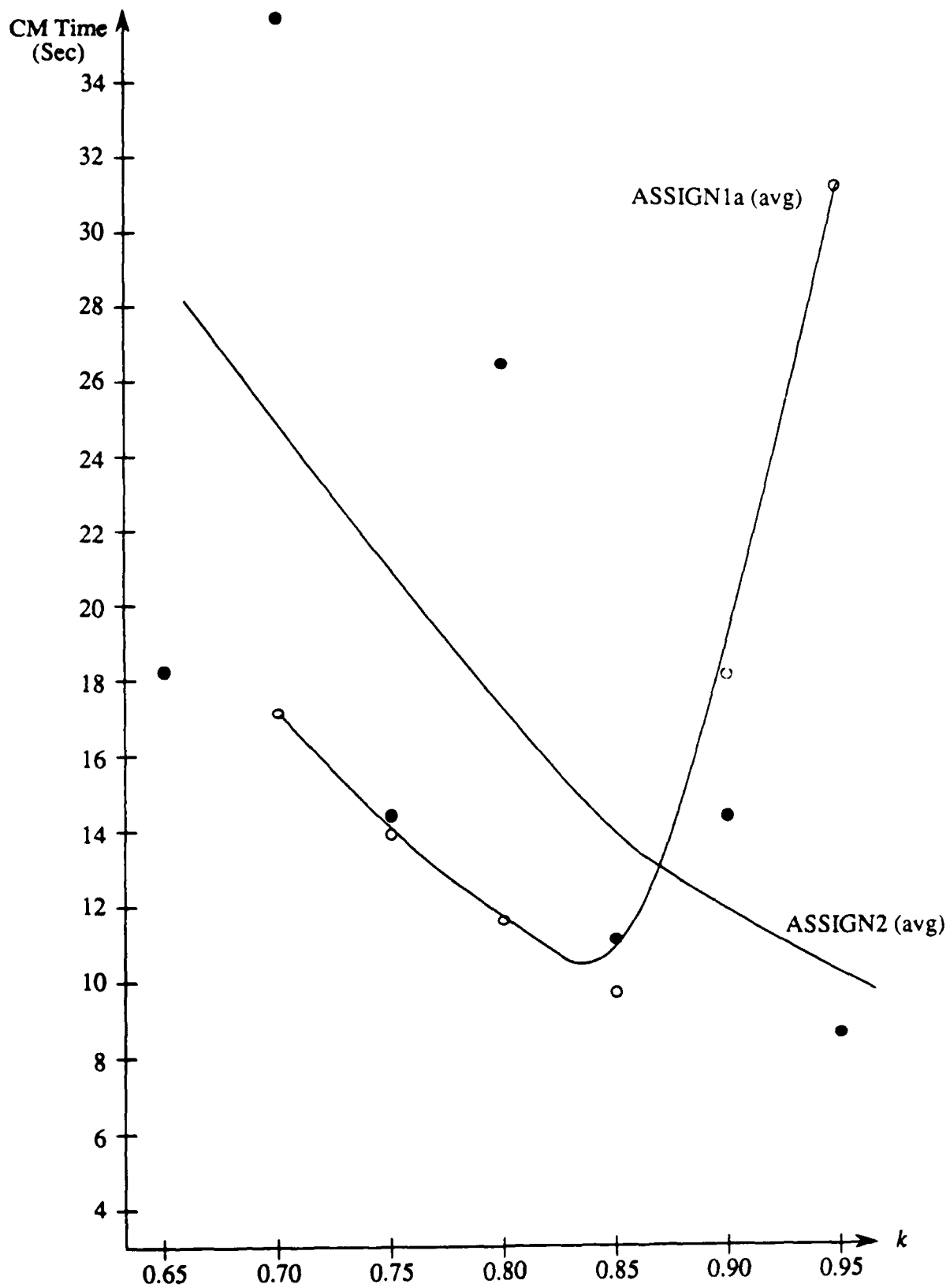


Figure 5-3: CM time (estimated for a 64K machine) for optimal assignments as a function of the tail-cutting parameter  $k$ . For problems with small cost range, such as ASSIGN1a (empty

We now look at the runs that were automatically terminated and investigate the quality of the partial assignment as a function of the parameter  $k$ . In part, this investigation will tell us how close the run was to terminating, although we know that the final sequential tail can be quite long. We really intend this study to be a preliminary investigation of the question we raised earlier about intentionally stopping short of optimal in the interest of saving time.

As we mentioned earlier, it is somewhat difficult to compare the incomplete runs given our termination criteria. Fortunately, for most of the problems, the best solution occurred early in the testing, and most of the incomplete runs were terminated after the same number of bidding iterations. In most cases this termination does not take effect until after **dec-factor** = 2.

Although we compare terminated runs based on the value of the partial assignment, we comment here briefly about running time. Most of the runs that were automatically terminated ran at approximately the same total and CM time for a given problem. Some runs required significantly less time even though they were cut off at the same number of iterations. To understand why, we consider the interaction between two of our optimizations: the tail-cutting parameter  $k$  and global bidding. By using the tail-cutting parameter  $k$  at a value  $k < 1$ , we always stop after  $p < n$  people are matched for all except the final auction round. Therefore, we always have more than one person bidding except during the final round. When there is only one person bidding, we can process the one bid faster using global operations. The faster runs are the ones that made it into the final "global bumping" stage where one person "bumps" another until the optimal solution is reached. The faster a run is relative to the typical time for that problem, the more time it spent in this final phase before automatic termination. Therefore, we are not surprised to notice that these faster-than-typical runs also produce the best partial assignments.

Figure 5-4 graphs the average percent error for problem ASSIGN2 (500 × 500, range 1-1000) as a function of the tail-cutting parameter  $k$ . The average for each  $k$  are taken over those values of **dec-factor** for which most runs were stopped early. For those runs which did complete for a value of **dec-factor** in the range of interest, we averaged in the optimal assignment value. That way we do not penalize a value of  $k$  for running to completion. Problem ASSIGN2 is a good example of the behavior for those problems that showed a dependence on  $k$ . They behaved similarly to the optimal runs just discussed. Namely, they performed better for higher values of  $k$  (in the neighborhood of 0.95). This improvement with increase in  $k$  probably occurs in part because more people are matched in the first auction rounds. This particular problem (ASSIGN2) was run for many more iterations than most and therefore we feel that it reflects the affect of termination when the algorithm is in or near its final tail. There was quite a bit of variance in some problems which showed this tendency, and for some problems a similar graph would appear quite flat, possibly because the runs had not, in general, proceeded far enough to show any tendencies.

We now look at the dependence of the running time upon the parameter **dec-factor**. One might think that the factor by which one divides  $\epsilon$  after each auction should have little affect upon the performance of the algorithm. In fact, it had a dramatic effect in the sense that for almost all problems with large value range (the ones that run longer in general), the best runs occurred at very low values of **dec-factor**. Furthermore, these runs were the best by far enough that few other runs even finished. For example, problems ASSIGN1c (value range 1-1000) and ASSIGN1d (value range 1-10,000) both had the overall best run at **dec-factor** = 2 and, aside from one run at **dec-factor** = 3, no other runs completed after twice as many iterations.

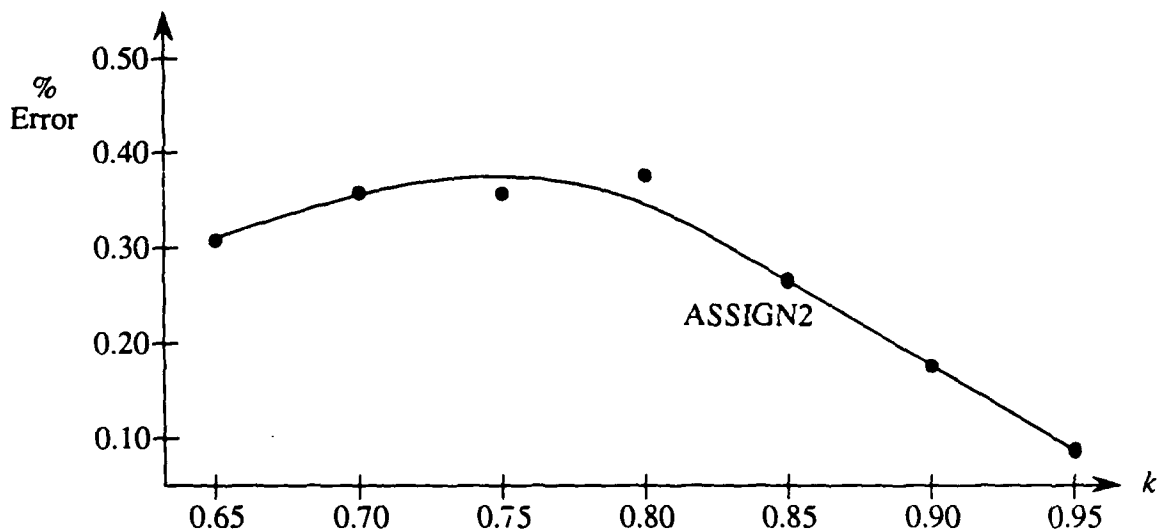


Figure 5-4: Average percent error of partial assignments as a function of parameter  $k$  for problem ASSIGN2 (500 × 500, range 1–1000). Higher values of  $k$  seem to work better and, in fact, the dependence is monotonic in the range  $k = 0.80 - 0.95$ .

We now look a little more closely at the dependence on **dec-factor** for those relatively few runs that finished. As illustrated in figure 5-5, for problems such as ASSIGN1a with the lowest value range, any value of **dec-factor** over 4 seemed to work about as well, better than lower values of **dec-factor**. This once again is in marked contrast to the problems with larger value range. Problem ASSIGN2, for example, performs well for **dec-factor**  $\leq 7$  and then times become dramatically worse.

We now look at the quality of the partial assignment as a function of **dec-factor** for those runs that were terminated early. We only have data for the problems with large value range because the smaller problems were monitored by hand. The dependence upon **dec-factor**, for problems with large value range, is much more pronounced than the dependence upon  $k$ . All of the test problems did better for lower values of **dec-factor**. Figure 5-6 illustrates the average percent error as a function of **dec-factor** for problem ASSIGN1d. The average is taken over  $k$ . The behavior is a very good example of the general case.

## 5.6 Summary and conclusions

In this section we summarize the results of our implementation of the assignment problem algorithm, speculate about the good choices of parameters  $k$  and **dec-factor**, and suggest directions for future testing.

The results with the assignment algorithm are very encouraging, though not yet conclusive. Assignment problems with 1M variables can be solved within 0.5 - 1 minute of CM time. The solution time can be brought down to a few seconds with a fully configured machine. As we showed in section 5.5 we can achieve significant savings in CM time for suitable choice of the parameters  $k$  and **dec-factor**. Unfortunately, the algorithm appears to be rather sensitive to

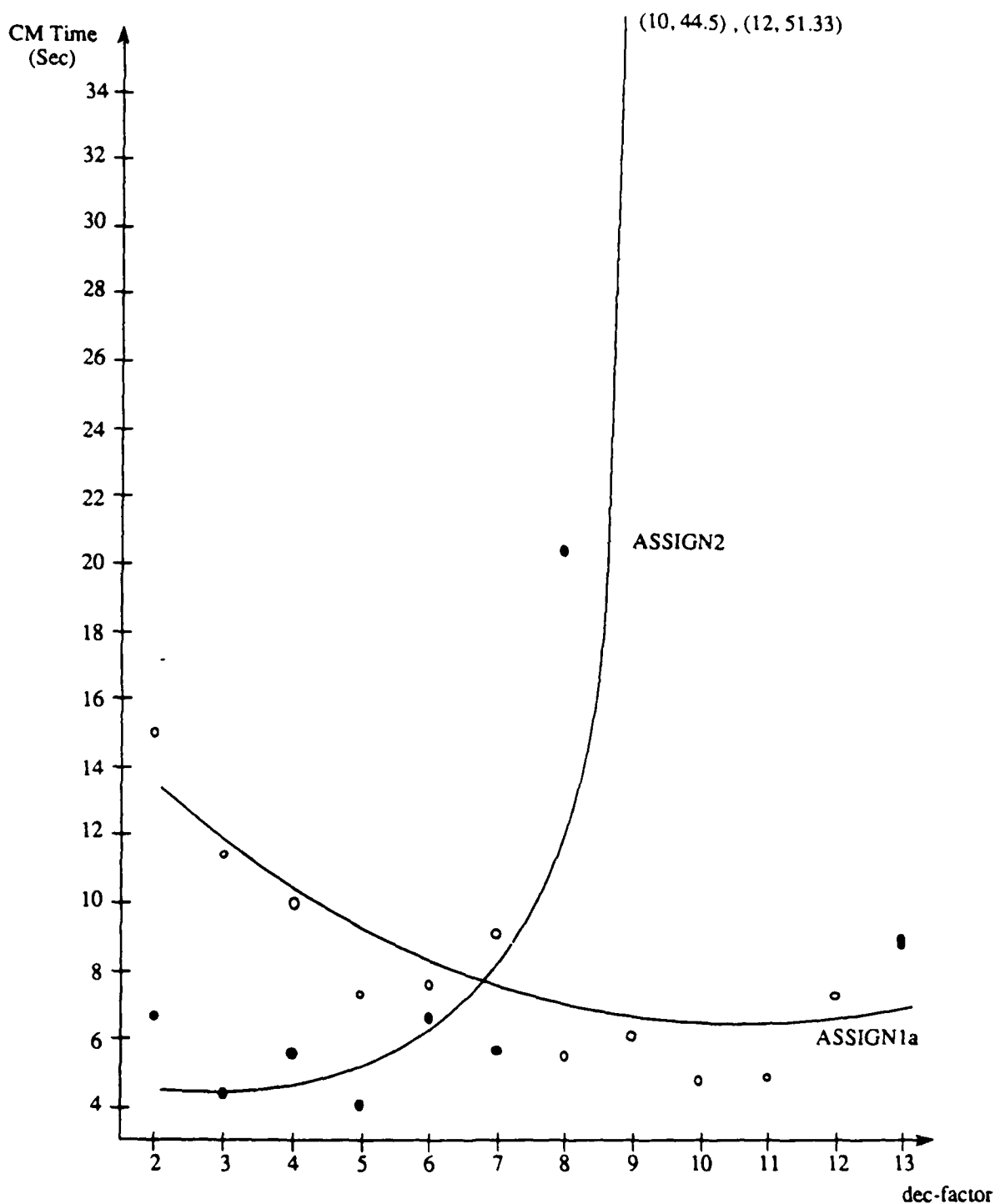


Figure 5-5: CM time (estimated for a 64K machine) for optimal assignments as a function of parameter **dec-factor**, which controls the number of auctions. We show best times rather than average since in general few problems completed other than at low values of **dec-factor**. Problems with small value ranges such as ASSIGN1a (empty circles) do better for moderate values of **dec-factor**. Those with larger value ranges such as ASSIGN2 (solid circles) do dramatically better at low values of **dec-factor**.

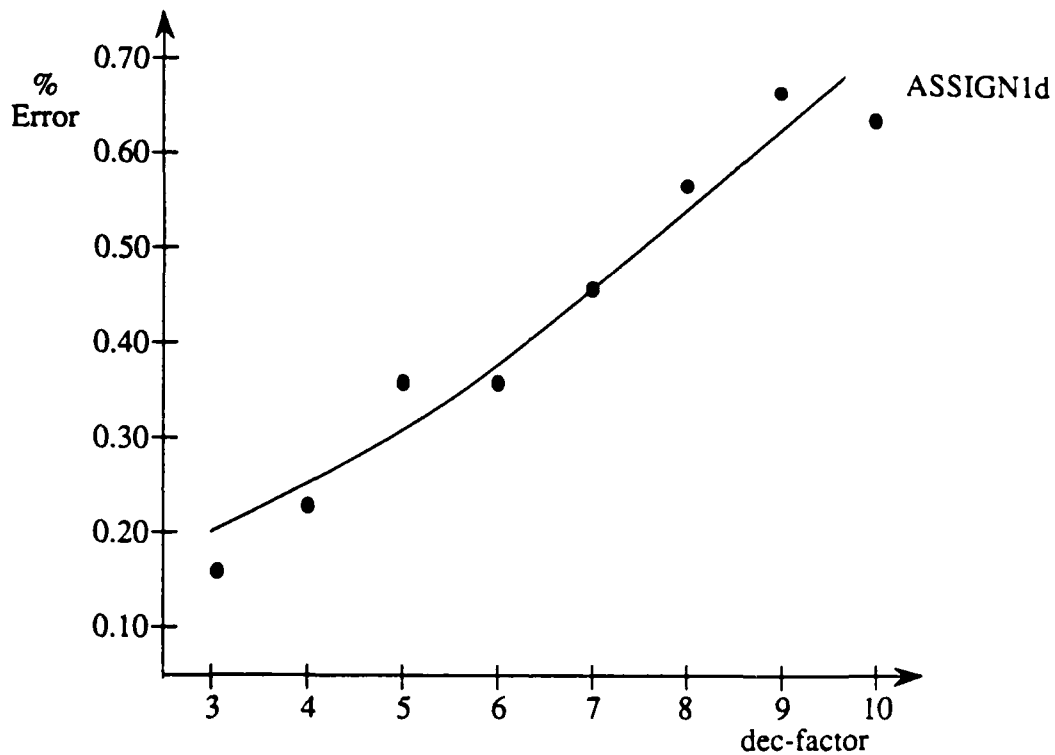


Figure 5-6: Average percent error of partial assignments as a function of parameter **dec-factor**, which controls the number of auctions. Because most runs completed for **dec-factor** = 2, one can assume another point at (2,0). For problems with large value ranges, the error increases almost monotonically with increasing values of **dec-factor**.

this choice and some fine tuning is required to achieve optimal performance. Additional work is required in order to more fully understand the behavior of the parallel auction algorithm and hence develop a set of settings for the parameters that are efficient for broad classes of problems. Such settings may depend upon general problem characteristics such as size and value range.

Based upon our limited testing, we make the following observations about the general behavior of the auction algorithm under various setting of the parameters. For the  $1000 \times 1000$  problems with value ranges 1-10, intermediate values of parameters **dec-factor** and  $k$  seemed to work best. Values of **dec-factor** in the range 4-7 coupled with  $k = 0.85$  and values of **dec-factor** in the range 8-11 coupled with  $k = 0.85$  or  $0.90$  consistently performed the best. For a given value of **dec-factor**, the fall-off in performance after the best value of  $k$  was dramatic. For problems with larger value ranges, the best performance seems to occur at low values of **dec-factor** such as 2 or 3 coupled with high values of  $k$  such as 0.9 and 0.95.

It seems that the algorithms performance is more sensitive to value range than to problem size (number of nodes). Unfortunately, we have only two data points for the latter comparison, namely problem ASSIGN2 vs. problem ASSIGN1c. Therefore we feel that there is insufficient data to allow us to draw conclusions about the effect of problem size with any degree of confidence.

As we mentioned at the start of section 5.5, we handicapped ourselves in our investigation of suboptimal assignments. We feel that the results are very good in light of this handicap, and that a more thorough study is warranted. In particular, we suggest running problems such as the ones we ran, closely monitoring the progress made in each iteration. It may be that only a few iterations almost always bring us within, say, 2 percent of the optimal solution. If this is true, then in the case where a close solution is sufficient, it will be very hard to beat the Connection Machine.

Even if we require an optimal assignment, we can benefit from this monitoring. We can run a problem for a predetermined (short) period of time for a variety of parameter settings that are expected to work well for the given value range. Alternatively, we can run simultaneous versions or dovetail. We can then determine the most promising runs, for example the ones that are in the final global bumping, and run one of those to completion.

## Chapter 6

# Parallel Traveling Salesman Heuristics

### 6.1 Introduction

The *traveling salesman problem* (TSP) is defined as follows: given  $n$  cities and the distances between each pair, construct a tour of the  $n$  cities of minimum distance. More formally, given  $n$  cities  $c_1, c_2, \dots, c_n$  and a value  $d(c_i, c_j)$  for all  $i \neq j$ , find a simple cycle  $T = \langle c_{i_1}, c_{i_2}, \dots, c_{i_n} \rangle$  which includes all  $n$  cities such that  $\sum_{\{c_i, c_k\} \in T} d(c_i, c_k)$  is minimized. In the *Euclidean* traveling salesman problem, each city has an  $x$  and  $y$  coordinate and the distance between cities is the standard Euclidean metric. We implemented parallel heuristics for the Euclidean traveling salesman problem on the Connection Machine CM-2. This chapter represents joint work with Ron Greenberg and Joel Wein.

Because we assume a complete graph on the  $n$  generated cities, the cost of assigning one processor per edge is prohibitive. For example, on the 16,384-processor CM-2 that we used, if we were to explicitly represent the  $[(2048)^2 = 4,194,304]$ -entry edge matrix generated by a 2048-city problem, each processor would have to simulate 256 processors. We ultimately would like to run heuristics on a million cities or more, so we cannot expect any machine available in the near future to have sufficient processors or memory to allow this explicit representation. In fact, many theoreticians agree that a parallel algorithm is "feasible" only if its processor requirement grows at most linearly with problem size. We chose, therefore, to assign one processor per city. We represent the tour by two parallel variables (pvars) containing the  $x$  and  $y$  coordinates of the city represented by that processor. The tour is formed by passing through the cities in processor order with wraparound. When we refer to the "edge" in a given processor, we mean the edge of the tour that originates at the city stored in that processor.

The cities were generated randomly and locally, using the Connection Machine random number generator to create integral  $x$  and  $y$  coordinates between 1 and  $n$ , the number of cities. The probability that a given pair of cities has identical coordinates is  $1/n^2$ . The probability  $P(M)$  that some pair of cities has identical coordinates is the probability that the second city matches the first, or the third matches one of the first two, and so on. Thus the probability of

a match is bounded by

$$\begin{aligned}
 P(M) &\leq \frac{1}{n^2} + \frac{2}{n^2} + \cdots + \frac{n-1}{n^2} \\
 &= \frac{n(n-1)}{2n^2} \\
 &= \frac{n-1}{2n}.
 \end{aligned}$$

This is a rather high probability of a match, but we are not concerned because none of the heuristics we implemented requires that the cities be distinct. We would hope that in the case that cities  $c_1$  and  $c_2$  are the same, that the heuristics will place them together in the tour yielding a 0-length edge. Thus we can model the case where the traveling salesman has two appointments in the same city.

The general strategy we adopted was to generate an initial tour of varying quality, and then “tweak” it. We investigate two types of techniques for generating an initial tour. The first, described in section 6.2.1, is *greedy* heuristics where we add cities to the tour one at a time. The second type of technique, described in section 6.2.2, is *partitioning* heuristics where cities are grouped into local tours which are then tied together to form the final tour. Tweaking, described in section 6.3, iteratively improves a tour by repeatedly exchanging pairs of edges. Refer to section 1.3 for details of the Connection Machine language primitives we use to describe the implementations of the heuristics.

Unlike the code for the matrix-vector primitives, we did not do any “speed hacking” on this code, even to the extent of using Paris (the CM version of assembly code). We aimed for readable, easily debugged high-level code that had no obvious inefficiencies. We did not intend this initial investigation to serve as a comparison of the raw performance of the Connection Machine versus, say, the Cray. Our goal was to develop parallel versions of known sequential heuristics as a first step toward the development of new parallel heuristics. Thus we investigate which heuristics map well onto the Connection Machine architecture for modest size problems, and try to hypothesize about which techniques will work best asymptotically in terms of both time and quality of tour.

We implemented our heuristics on a 16K (16,384)-processor CM-2 running with a 4MHz clock. The results of our limited testing are presented in section 6.4. The algorithms for generating the initial tour perform a set number of tasks per city or group of cities. Therefore, for these heuristics, the running times grow predictably with the problem size  $n$ . Tweaking, however, is performed iteratively until a termination criterion is met, and therefore the running time is unpredictable. We found that most of the improvement from tweaking was achieved in early iterations and the final few percentage points of gain, the *tail*, accounted for a disproportionately large amount of compute time. Because we are running heuristics for an NP-complete problem, we do not expect to arrive at the optimal tour, and therefore we can stop the tweaking process a little short of its ultimate gain to save time. This is in contrast to our implementation of the assignment problem discussed in chapter 5. There we cut the sequential tail in intermediate iterations, but we had to run the final tail to the end to guarantee an optimal solution. For the TSP heuristics, we found that cutting the tail in the tweaking procedure (that is, stopping after a certain percentage improvement over the initial tour) cut the running time substantially without substantially decreasing the quality of the tour.

## 6.2 Building the Initial Tour

This section describes methods for generating an initial tour from the randomly generated cities. We discuss some of the existing sequential heuristics for TSP that we did not use and why. We then discuss the greedy and partitioning heuristics that we did implement.

Once one assumes a triangle inequality on the set of distances between the cities, there are several known heuristics that are guaranteed to produce a tour of length at worst some small constant factor longer than the optimal tour length [9,64,105]. One standard approach is based on the construction of a minimum spanning tree (MST). Given a depth-first traversal of such a tree one can "shortcut" around the repeated nodes to construct a tour of all the cities. This tour is no more than twice as long as the optimal tour.

Christofides improved this result by constructing a minimum spanning tree and then a minimum weight matching on the odd-degree nodes. He then constructs an Euler tour of this graph and shortcuts it to a tour. The resulting tour length is no worse than  $3/2$  the optimal length.

These techniques seem well suited for parallelization, since building minimum spanning trees and min-weight matchings is possible in polylogarithmic time. The minimum spanning tree methods prove not to be useful for us, however, since they assume one processor per edge. Working only with a representation of the city positions, we saw no way to develop a polylogarithmic time algorithm for the minimum spanning tree. Essentially the troublesome step reduces to "Given  $n$  processors each representing a point in space, find the closest point to each point." This step appears to require  $\Omega(n^2)$  work which would imply that it cannot be completed in polylog time on  $O(n)$  processors. We can, of course, gain a factor of  $n$  from parallelism.

Given this situation, one can either use  $O(n)$ -time heuristics, or look for other polylogarithmic techniques. We pursued a small subset of each.

The simplest TSP heuristics are "greedy" heuristics, that at each step choose the city that is "best" in some sense — closest city, lowest cost inserted city, etc. [9,105]. In the nearest neighbor heuristic, for example, at each step the city closest to the last city on the tour is added. This heuristic is easily implemented in  $O(n)$  parallel time. Rosenkrantz, Stearns, and Lewis proved that in the worst case the ratio of the lengths of nearest-neighbor to optimal tours can grow as  $\Omega(\lg n)$  and that in fact this rate can be achieved [105]. The experimental results of Bentley and Saxe [9], however, indicate that quite often the results of the Nearest Neighbor Algorithm are comparable to or better than those of the MST heuristic. This experimental performance, combined with their ease of implementation, motivated our investigation of greedy techniques. Section 6.2.1 describes the set of greedy techniques we implemented.

A second type of TSP heuristic is a *partitioning* heuristic that divides cities into groups, forms a local tour, and joins the local tours to form a global tour. Section 6.2.2 discusses partitioning heuristics in more detail. In particular, it describes the *Strip tour* which is a partitioning heuristic that we implemented.

### 6.2.1 Greedy Heuristics

All of the greedy heuristics can be described within the following general framework:

0. Start the tour at an arbitrary city.

1. Choose a city to add to the tour.
2. Choose a position in the tour at which to add the city.
3. Add the city, and return to step 1.

For this reason, most of the greedy heuristics are performed by a single \*lisp function which calls subroutines to choose the cities and positions and make the insertions. The exception is the nearest neighbor-heuristic, which was coded separately since it requires significantly less work to perform the relevant subfunctions.

The simplest of the greedy heuristics is the nearest-neighbor heuristic. This heuristic starts the tour at an arbitrary city and performs  $n - 1$  iterations of adding to the end of the tour the city nearest the last one visited. This is straightforward to implement both sequentially and in parallel. The sequential running time is  $O(n^2)$ . In parallel, it is  $O(n)$  because we can determine the next city to add in constant time. We simply broadcast the coordinates of the last city, compute in parallel the distance from each city not on the tour to that city, and do a global minimum. Because of its simplicity, the constant in the running time is smaller than for the other heuristics to be discussed. We noticed that although the nearest-neighbor heuristic does not generally provide as short a tour as some of the other greedy heuristics, the resulting tour responds well to tweaking.

The other heuristics can be classified as insertion or addition heuristics. Most are discussed in Rosenkrantz, Stearns, and Lewis [105]; two are obvious extensions. For each of these methods, we choose a city  $c$  which is best in some sense relative to the tour constructed so far. We then choose a place to *insert* the city, meaning that it is added to the tour between two currently neighboring cities  $a$  and  $b$ . Thus the salesman now goes from city  $a$  to  $c$  to  $b$  instead of from  $a$  to  $b$ . The rest of the tour is unchanged. The *cost* of adding city  $c$  to the tour between cities  $a$  and  $b$  is the difference between the length of the new tour and the length of the old tour, namely  $d(a, c) + d(c, b) - d(a, b)$ .

We use four different methods for choosing the next city to insert. We define the *distance* of a city  $c$  to a partial tour  $T$  as the minimum of all the distances to cities in the tour:  $\min_{c_i \in T} d(c, c_i)$ . For the nearest-insertion and nearest-addition methods, we choose the city which is closest in distance to the tour constructed so far. For the farthest-insertion and farthest-addition methods, we choose the city which is farthest in distance from the tour. For the cheapest-insertion method, we choose the city for which we incur the least cost by adding it to the tour in its best place. For the random-insertion method, we just pick the next random city not in the tour.

There are two ways of deciding where to insert the selected city into the tour, which is what creates the distinction between insertion and addition heuristics. In insertion heuristics, the chosen city is inserted in the position in the tour which incurs the least cost, while in addition heuristics the city is added at the position which was responsible for making it best. For example, in nearest-addition, we add the closest city to the tour immediately after the city in the tour which is closest to it (even though adding in another position could be less costly).

All of the insertion and addition heuristics can be implemented in  $O(n^2)$  time sequentially and  $O(n)$  time in parallel. We use a technique mentioned in Rosenkrantz, Stearns, and Lewis with regard to nearest-insertion. Instead of recomputing the distance of all cities to the tour after each insertion, each city keeps track of its distance to the tour and performs a constant

time update by checking its old distance against its distance to the most recently inserted city. Rosenkrantz, Stearns, and Lewis recognized only an  $O(n^2 \lg n)$  (sequential) algorithm for cheapest-insertion, but, in fact, the same technique is applicable to cheapest-insertion.

There is only one implementation detail of the insertion and addition heuristics which is at all worth mentioning, since the translation from the descriptions above to \*lisp is quite straightforward. Instead of shifting the city data down the sequence of processors in order to accommodate the insertion of a city, each processor (city) simply keeps track of what position it belongs in as the tour is constructed. Then at the end, a single routing operation puts the cities in the desired tour order.

### 6.2.2 Partitioning Heuristics

In situations where the data points are uniformly distributed in the plane, more local techniques for initial-tour building can provide expected polylogarithmic performance. In this section we describe a two partitioning heuristics: the strip tour which we implemented, and Karp's partitioning algorithm, which we did not implement.

To apply the *strip tour* heuristic, we assume the cities are distributed in an  $n \times n$  square. We partition the square into  $\sqrt{n}$  vertical strips, either letting each strip contain  $\sqrt{n}$  points (*equipopulated strips*), or letting each strip be of physical width  $\sqrt{n}$ . We then sort the points in each of these strips by  $y$  coordinates. The strip tour is the result of starting at the bottom left hand corner, and constructing the tour that goes up the first strip then to and down the second, up the third, etc. The last point is joined to the first by one long edge.

Building this tour is quite simple on the CM. We start with an initial random tour in two pvars, one for  $x$  coordinates, one for  $y$  coordinates and proceed as follows:

1. Sort the tour by  $x$  coordinates. (Be sure to take the  $y$  coordinates to the right places)
2. Set up a boolean segment pvar to mark the boundaries of the strips.
3. Do a segmented sort, i.e. sort in each strip by  $y$ -coordinates. (An easy way to accomplish this is to add  $n * i$  to each value, where  $i$  is the segment number, and then do a regular sort.)
4. "Flip" the odd numbered segments to set up the tour order so that we are traversing down those segments.

What could ultimately be the most promising method of initial tour generation is Karp's partitioning algorithm [68]. This algorithm can roughly be understood as follows:

1. Divide the original area into  $2^k$  subrectangles.
2. Construct an optimal tour in each subrectangle.
3. Combine these local tours through shortcutting.

This algorithm can be implemented in  $O(T \lg N)$  time, where  $T$  is the time to construct each local tour. If we were to partition in such a way as to expect  $O(\lg N)$  cities per square, constructing the exact local tour in each square would take us back to expected polynomial time. If we were again to use a heuristic, such as nearest neighbor, to construct the local tours,

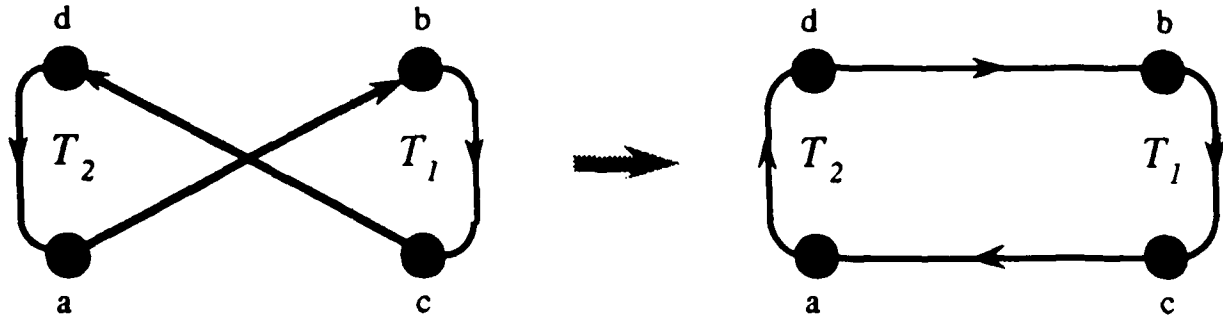


Figure 6-1: We tweak a pair of edges by replacing them by two edges, one that joins the heads and one that joins the tails. Half the tour is then reversed to yield a new directed tour. A tweak is advantageous if by performing the tweak we reduce the length of the tour.

we would have a polylogarithmic heuristic that we expect would be more accurate than the strip tour. This heuristic has not been implemented although implementing it should not be particularly difficult.

### 6.3 Tweaking

In this section we discuss the implementation of tweaking heuristics. We first define the tweaking operation. We then discuss three implementations of tweaking. The first implementation finds tweaks in parallel but performs them sequentially. The other two implementations perform tweaks in parallel.

Consider the quadrilateral defined by four points in the plane, for example points  $a$ ,  $b$ ,  $c$ , and  $d$  in figure 6-1. The *quadrangle inequality* states that the sum of the lengths of either set of opposite sides is less than the sum of the lengths of the two diagonals. For example, in figure 6-1, we have that  $d(a,d) + d(b,c) < d(a,b) + d(c,d)$ , and similarly  $d(d,b) + d(a,c) < d(a,b) + d(c,d)$ .

When we apply the quadrangle inequality to the traveling salesman problem, it means that in the optimal tour, no edge crosses back on the tour. More specifically, suppose we are given some directed tour of the cities such as the tour illustrated in figure 6-1. In this tour, it is better to replace the long pair of crossing edges  $(a,b)$  and  $(d,c)$  with the edges  $(c,a)$  and  $(d,b)$ . The direction of the directed path  $T_2$  must be reversed to maintain a legal tour. We call such an edge replacement with path reversal a *tweak*. A tweak is *advantageous* if by performing the tweak we reduce the length of the tour. If no pair of edges can be tweaked advantageously, we say the tour is *2-optimal*.

We repeatedly perform advantageous tweaks until one of two termination criterion is satisfied. If the user furnishes an optional parameter indicating the desired percentage improvement  $p$ , then tweaking stops when the tour is  $p$  percent shorter than the initial tour or when the tour is 2-optimal. Otherwise, if no percent-improvement goal is specified, the tour is tweaked until it is 2-optimal. If allowed to run to 2-optimality, the tweaking procedure frequently has a long *tail* during which it spends a lot of time finding tweaks, and the ones it finds are of steadily decreasing value. We can cut the tail by specifying an appropriate percent-improvement goal.

One might at first believe that by considering the tour undirected, allowing traversal in

either order, path reversal in tweaks is unnecessary. If the tour in figure 6-1 were undirected, however, then we could not distinguish between the desired edge pair  $(c, a)$ ,  $(d, b)$  and the edge pair  $(a, d)$ ,  $(c, b)$  which shortens the tour by chopping it into two smaller disjoint tours. In a directed tour, to tweak edges  $(tail1, head1)$  and  $(tail2, head2)$ , we replace the two edges with  $(head1, head2)$  and  $(tail1, tail2)$ , and then reverse the path originally going from  $head1$  to  $tail2$ .

In our tour representation, a tweak consists of flipping a set of consecutive entries in the city pvar. For example, to perform the tweak illustrated in figure 6-1, we take the piece of the tour pvar from processor  $d$  to processor  $a$  inclusive and reverse it. Alternatively, we can reverse from processor  $c$  to processor  $b$  inclusive to get the directed tour going in the opposite direction. Thus we can always tweak by flipping a consecutive set of cities in the pvar representation without worry of wraparound. To actually perform the reversal, we broadcast the addresses of the processors at the ends of the path to be reversed. All cities then determine if they must move and if so what processor they must move to. Then all cities ship themselves off if necessary with a \*pset.

The choice of edge pair to tweak is also heuristic. Initially, at least, it seems that tweaking the longest edges will maximize our gain. Using this assumption, in our simplest, "global" tweaking routine, we chose an edge pair as follows:

1. Make all edges in the tour active. If edge  $(x, y)$  is in the tour, then the processor containing city  $x$  controls edge  $(x, y)$ .
2. Find the edge  $(a, b)$  of maximum length quickly using the wired OR (\*max).
3. Broadcast  $(a, b)$ .
4. Each city  $c$  which is followed by city  $d$  in the tour calculates the incremental change to the tour caused by tweaking  $(a, b)$  and  $(c, d)$ .
5. Choose the most advantageous tweak using the \*max function. If the best tweak is advantageous, perform that tweak and go back to step 1. If no tweak is advantageous, inactivate edge  $(a, b)$ , and go back to step 2.

The tour is 2-optimal if no active edges remain. The only way we can see to deterministically find the tweak with the maximum gain in constant time is to use  $\Omega(n^2)$  processors and check each pair of edges in parallel. Unfortunately the processor cost is prohibitive. Using  $O(n)$  processors, we can determine the best tweak in  $\Theta(n)$  time, but we feel it is better to do  $\Theta(n)$  good tweaks in that time than one great tweak.

We implemented two versions of the tweaking heuristic that allow limited parallelization of tweaking. It is not obvious how one goes about parallelizing the tweaks. For example, how does one calculate the tour resulting from parallel overlapping reversals and how does one efficiently choose in parallel a set of good tweaks? We chose the simplest form of parallelization, namely we divided the tour into disjoint sets of consecutive cities and performed the tweaking procedure in parallel on each set of cities. For each group of cities, we find the maximum-length active edge and perform the most advantageous tweak with that edge, if any.

For this simple division, tweaks do not interfere. Suppose in the current tour the cities are ordered  $t_1, t_2, \dots, t_n$ . If we divide into groups of size  $g$ , then the first group consists of

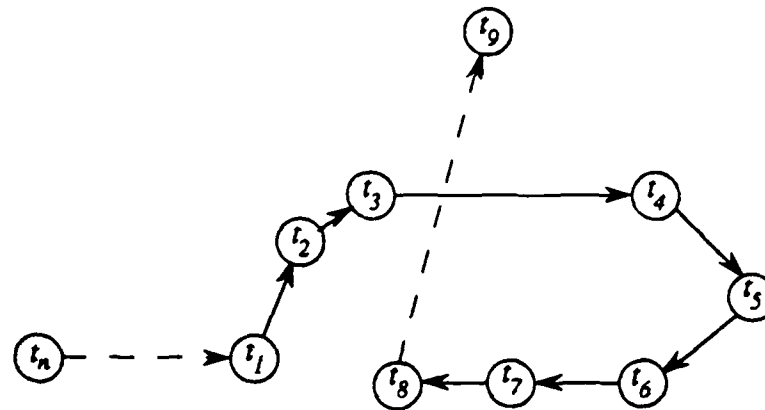


Figure 6-2: In a parallel tweak, we perform tweaks within contiguous groups of cities on the tour. The edges considered for tweaking are the tour edges originating at each city in the group. Dashed edges indicate tour edges that go across group boundaries. In this case the dashed edge  $(t_8, t_9)$  is involved in an advantageous tweak which can be performed by reversing cities  $t_4$  through  $t_8$  inclusive.

cities  $t_1, t_2, \dots, t_g$  and tour edges  $(t_1, t_2), (t_2, t_3), \dots, (t_g, t_{g+1})$ . That is, we do not consider the groups to be tours themselves with an edge wrapped back. This assumption could lead to incorrect tweaking decisions, especially since the false tie-back edge could be quite long. Instead we allow tweaking on the tour edge that leaves the last processor in the group. If this edge is chosen for a tweak, we can still correctly perform the tweak by reversing a set of cities strictly within that group. For example, in figure 6-2, we show a group of size 8. The dashed lines indicate tour edges that go across group boundaries. In this case it is advantageous to tweak edges  $(t_3, t_4)$  and  $(t_8, t_9)$ . We can do this by reversing cities  $t_4$  through  $t_8$  inclusive, which is strictly within the group.

In a parallel tweak, each group performs the best tweak that it found in the last iteration of the tweak-finding procedure, provided that the tweak is advantageous. We only perform parallel tweaks when at least one group has found an advantageous tweak.

By dividing into groups, we can determine and perform multiple advantageous tweaks at the same time, but each iteration runs more slowly. Each global wired-OR operation (such as  $\text{*max}$ ) and each broadcast is replaced by a segmented scan. We estimated that a parallel iteration runs about 32 times slower than a global iteration, so we never perform parallel tweaking on less than 32 groups.

Our first attempt at parallel tweaking is a one-pass version. We divide the cities into groups of 4, and perform tweaks in parallel on each group until all groups are 2-optimal. We then double the group size and repeat the process until there are less than 32 groups. We then do a straight global 2-optimization. As with the global tweaking routine, all parallelized versions will accept an optional argument specifying a satisfactory percentage improvement and the routine will terminate if the tour achieves the desired percentage improvement.

We also implemented a parallel tweaking procedure that operates in multiple passes. For the multiple-pass version, we start by performing some tweaks at the top level. Then we divide the cities into 32 groups of size  $n/32$  and perform some number of parallel tweaks at that level.

Then we halve the group size, and so on until the groups are of size 2. We then go back to the global level and repeat the process with the groups shifted. If the groups are of size  $g$ , then we tweak until all the groups are 2-optimal or until we perform  $g/4$  parallel tweaks, or until we have achieved the desired percent improvement if specified. If we are trying to make the tour 2-optimal, the procedure must end at the top level (ie. the global tweaking). In the best case, we perform  $n/4$  tweaks at each level so we achieve  $\Theta(n \lg n)$  tweaks in  $\Theta(n)$  time. The overall time is dominated by the time spent at the global level since the time spent on each level is roughly geometrically decreasing, and yet we can possibly perform the same number of tweaks on each level. We chose to go from global tweaks to tweaks at smaller levels because we envisioned that the globally best tweaks would be performed first, bringing cities closer to where they would ultimately be, and then local tweaks would act as fine-tuning.

## 6.4 Results and Conclusions

In this section we attempt to draw some conclusions about the best strategies for achieving near-optimal solutions to the Euclidean TSP on the Connection Machine. We present the results of our limited testing in tabular form. We then suggest possible explanations for the results we achieved and speculate about which methods might be best asymptotically in terms of both time and final tour length. All of our conclusions about performance of initial tour methods, etc. are implicitly prefaced by the statement "for a uniformly distributed set of points ..." We did not implement any other distributions of random initial tours.

Our results appear in tables at the end of this chapter. We performed all experiments on a 16K (16,384)-processor CM-2 running with a 4MHz clock. Tables 6.1 and 6.2 present the results of all of our basic methods on test cases of 128 and 256 cities respectively. More specifically, we apply every initial tour method and then tweak them to 2-optimality using the global tweaking (tweaks performed sequentially). The results are presented in increasing order of final tour length. The length of the random tour given in the first row of each table is the value of the tour implied by the initial randomly generated pvar of cities.

Next we selected the most promising initial tour methods, nearest neighbor and nearest addition, for testing on larger problems with more sophisticated tweaking approaches. Tables 6.3 and 6.4 list the results of one-pass and multiple-pass 2-optimal tweaking, as well as parallel tweaking to a given goal percentage improvement, on test cases of 256 and 2048 cities respectively. Finally tables 6.5 and 6.6 list the results of strip tours performed on worlds of 16k and 64k cities respectively.

As we mentioned before, when we implemented these heuristics, we were not trying to achieve raw performance, but were investigating relative performance of the various techniques when implemented on the Connection Machine architecture. Nevertheless, we feel that it is only fair to mention that since these figures were gathered, the CM-2 has been improved. The current CM-2 has faster system software, a faster and better random number generator, faster clocking, and more memory. Thus, even if we were to run the same code today, our times would be faster than those listed in the tables.

The features of the Connection Machines that proved most valuable were the fast max, min, and broadcast, and the scanning facility. Due to our sparse problem representation, chosen to allow us to attack large problems, the advantage of the Connection Machine often consisted primarily of a factor of  $n$  improvement resulting from the fast max, min, and broadcast

operations. In the construction of strip tours and the attempts at parallel tweaking, the scanning and routing facilities of the Connection Machine were used more fully.

We now comment upon the general performance of the various greedy heuristics. Nearest-neighbor is fastest, as would be expected. The addition methods take about 3 times as long as nearest-neighbor, and the insertion methods take about 6 or 7 times as long. Since the addition methods generally yield the longest tours, they seem to offer no advantage over use of the nearest-neighbor heuristic. Only if they were generally more amenable to tweaking, would they be useful. We did encounter one example of this behavior among our test cases, but this is probably not true in general. The greedy heuristic which yields the shortest tour is almost always farthest-insertion (surprisingly, better even than cheapest-insertion). These farthest-insertion tours were nearly 2-optimal and quite good even compared to 2-opted versions of the other methods. As the problem size grew, however, we were able to beat farthest-insertion in both time and tour length using parallel tweaking with a suitable goal percentage starting from a faster initial tour construction method.

The strip tour requires two \*sort operations and one additional reasonably local routing operation. The time of execution is accordingly extremely fast. As we see from table 6.6, on a 16384-processor machine we constructed a strip tour for a 65,536 city world in .99 seconds.

Although the length of the strip tour is bounded above by  $2n\sqrt{n}$  for an  $n \times n$  square, it performs significantly worse than the greedy heuristics. Thus the time gained in the initial tour construction is lost in tweaking this worse initial tour to two-optimality.

Tweaking a tour to 2-optimality required much less time if the tour was already nearly 2-optimal. Tweaking a random tour required at least twice as much time as tweaking a tour built by one of the initial tour heuristics. Of course we expected much poorer behavior on inferior initial tours because many more tweaks are required to reach 2-optimality. If we are actually tweaking to full 2-optimality (no goal percentage), we spend  $\Theta(n)$  time just to verify the termination criterion. By the final stages, when we are performing mostly local tweaking, we spend  $\Theta(n)$  time finding a pair to tweak, since most of the longest-length edges have stabilized. In practice, we observed about  $O(n^2)$  behavior for 2-opting.

As seen from the relatively small running times of tweaking with percentage goals, It appears that most of the gain from tweaking occurs early in the procedure. Unfortunately, it is quite difficult to decide upon a reasonable goal percentage. The percentage improvement depends upon the quality of the initial tour. Also, different initial tours of similar quality can lead to different percent improvement depending upon which local minimum the tweaking routine arrives at. If the user is too pessimistic, the routine will run very fast, but the improvement will not be anywhere near as good as it could be. If the user is too optimistic, the routine may end up doing straight 2-opting (ie. tweaking until the tour is 2-optimal). The situation is particularly delicate since, according to our limited testing, the fall-off in gain vs. time is very steep near the ultimate percent gain, but the cost of additional percentage gain is very small even up to, say, 2 percent off the ultimate gain. For example, as seen in table 6.4, running multiple-pass tweaking on a nearest-neighbor tour of 2048 cities with a 19% cutoff resulted in a tour that was 0.002 percent worse than the full 2-opted multiple-pass version, and it ran in less than half the time. Of course the user does not want to 2-opt in order to determine a good goal percentage, so we conclude that it might be better to monitor the gain of the system and stop when the average gain over a "reasonable" period of time is "small", or until the search for an advantageous tweak requires, say,  $\Theta(n)$  time.

As the size of the problem grew, the multiple-pass tweaking seemed to perform the best of the three tweaking routines. We feel that the multiple-pass tweaking could be made more efficient by modifying the termination criteria as described above, and perhaps by reducing the time spent on each level. For the relatively small problems that we tested, the routine did not perform many passes. That is, the routine only went from top-level down to small groups two or three times. We would have to experiment to determine a good trade-off between number of passes and time spent in overhead for the transitions between computing phases. With so few passes, we pay much more for "just missing" 2-optimality at the top level. We must complete the whole pass before we recognize that we are done, and with so few passes, the relative time wasted is large.

It seems that for medium-sized problems of a few hundred to a few thousand cities the most promising of the approaches we tried is the nearest-neighbor initial tour followed by multiple-pass parallel tweaking. As the number of cities grows, however, we expect that the  $\Theta(n)$ -time greedy heuristics and global 2-opting will become prohibitively expensive. The blazing speed of the strip tour leads us to believe that for large problems we will benefit from a similar expected log-time heuristic such as the Karp partitioning method. Hopefully, such a method not only would be very fast but also would produce much better tours than the strip method and be amenable to parallel tweaking.

Another possible optimization is the Lin, Kernighan generalization of tweaking called  $k$ -opting [84]. Their algorithm exchanges  $k$  edges of the tour, determining both  $k$  and the edge-sets to exchange on the fly. We anticipate that the time spent determining a  $k$ -tweak will increase substantially over the time to find a 2-tweak, but the number of tweaks may also diminish substantially.

		128 Cities					
Tour Type	Length	Initial Time		Tweak Time		Total Time	
		Total	CM%	Total	Cm%	Total	CM%
Random	8302.78						
2-Opt Farthest Addition	1177.53	3.37	33.22	29.08	35.41	32.45	35.18
2-Opt Nearest Addition	1183.01	3.34	33.26	12.58	35.25	15.92	34.83
2-Opt	1217.80			56.50	35.24	56.5	35.24
2-Opt Random Insertion	1222.72	6.38	47.48	13.13	35.20	19.51	39.22
2-Opt Farthest Insertion	1222.89	7.72	44.83	11.22	35.91	18.94	39.55
2-Opt Nearest Neighbor	1224.24	1.14	35.67	12.34	35.24	13.48	35.28
2-Opt Cheapest Insertion	1237.97	7.44	45.08	14.18	35.93	21.62	39.08
Farthest Insertion	1242.68	7.72	44.83			7.72	44.83
Random Insertion	1263.58	6.38	47.48			6.38	47.48
2-Opt Nearest Insertion	1278.45	7.74	44.64	12.76	35.50	20.50	38.95
Cheapest Insertion	1326.71	7.44	45.08			7.44	45.08
Nearest Insertion	1379.69	7.74	44.64			7.74	44.64
Nearest Neighbor	1483.75	1.14	35.67			1.14	35.67
Nearest Addition	1592.58	3.34	33.26			3.34	33.26
Farthest Addition	2255.46	3.37	33.22			3.37	33.22

Table 6.1: Time is measured in seconds. Methods are ordered by increasing length of the tour they produced. The value of the Random Tour in the first line is the value implied by the pvar of randomly generated cities. All heuristics start from this tour.

256 Cities / Global Tweak							
Tour Type	Length	Initial Time		Tweak Time		Total Time	
		Total	CM%	Total	Cm%	Total	CM%
Random	34790.28						
2-Opt Nearest Neighb	3174.96	2.39	36.33	47.70	34.74	50.09	34.82
2-Opt Random Insert	3271.24	12.84	47.46	48.39	35.16	61.23	37.74
2-Opt Nearest Add	3282.56	6.70	33.82	61.34	34.76	68.04	34.67
2-Opt Farthest Insert	3298.05	15.38	44.94	27.50	34.84	42.88	38.46
Farthest Insertion	3312.30	15.37	44.94			15.37	44.94
2-Opt Farthest Add	3335.67	6.69	33.40	144.70	34.66	151.39	34.60
2-Opt Cheapest Insert	3347.32	14.76	45.35	81.84	34.70	96.60	36.33
2-Opt Strip	3349.41	0.20	47.61	94.27	35.14	94.47	35.16
2-Opt	3364.39			152.76	34.78	152.76	34.78
Random Insertion	3384.00	12.84	47.46			12.84	47.46
2-Opt Nearest Insert	3428.06	15.38	45.08	79.22	34.87	94.60	36.53
Cheapest Insertion	3731.10	14.76	45.35			14.76	45.35
Nearest Insertion	3886.22	15.38	45.08			15.38	45.08
Nearest Neighbor	3931.28	2.39	36.33			2.39	36.33
Nearest Addition	4312.01	6.70	33.82			6.70	33.82
Strip	4657.52	0.20	47.61			0.20	47.61
Farthest Addition	6138.93	6.69	33.40			6.69	33.40

Table 6.2: Time is measured in seconds. Methods are ordered by increasing length of the tour they produced. The value of the Random Tour in the first line is the value implied by the pvar of randomly generated cities. All heuristics start from this tour.

256 Cities / Parallel Tweak							
Tour Type	Length	Initial Time		Tweak Time		Total Time	
		Total	CM%	Total	Cm%	Total	CM%
Random	34790.28						
1-P-P NN	3173.74	2.40	36.33	43.77	35.08	46.17	35.14
M-P-P NN	3174.96	2.39	36.33	47.62	35.00	50.01	35.06
1-P-P 15% NN	3330.96	2.39	36.33	4.51	35.50	6.90	35.79
1-P-P 18% NN	3220.95	2.39	36.33	9.03	35.29	11.42	35.51
1-P-P 19% NN	3181.94	2.39	36.33	18.20	34.41	20.59	34.63
1-P-P Strip	3365.47	0.20	47.16	65.56	35.25	65.76	35.29
M-P-P Strip	3349.41	0.20	47.16	56.64	43.94	56.84	43.95
M-P-P 25% Strip	3474.75	0.20	47.16	13.77	33.80	13.97	33.99
M-P-P 27% Strip	3390.75	0.20	47.16	18.02	34.27	18.22	34.41
1-P-P 27% Strip	3391.98	0.20	47.16	31.09	35.12	31.29	35.20
M-P-P NA	3282.93	6.70	33.82	40.59	34.54	47.29	34.44
1-P-P NA	3317.69	6.70	33.82	49.85	32.92	56.55	33.03
M-P-P 20% NA	2448.65	6.70	33.82	11.01	33.67	17.71	33.73
M-P-P 22% NA	3357.16	6.70	33.82	21.46	34.45	28.16	34.30

Table 6.3: Time is measured in seconds. The value of the Random Tour in the first line is the value implied by the pvar of randomly generated cities. All heuristics start from this tour. The abbreviations should be interpreted as follows: 1-P-P = One Pass Parallel tweaking, M-P-P = Multiple Pass Parallel Tweaking, NN = Nearest Neighbor, and NA = Nearest Addition. Percent Figures indicate the tweaking was run until the specified percent improvement was achieved over the initial tour.

2048 Cities / Parallel Tweak							
Tour Type	Length	Initial Time		Tweak Time		Total Time	
		Total	CM%	Total	CM%	Total	CM%
Random	2158810.30						
Farthest Insertion	74873.89	123.86	45.11			123.86	45.11
Nearest Neighbor	84210.66	19.26	36.07			19.26	36.07
M-P-P 10% NN	75782.14	19.26	36.07	10.29	33.18	29.55	35.06
1-P-P 10% NN	75757.61	19.26	36.07	39.12	37.59	58.38	37.08
M-P-P 12% NN		19.26	36.07	24.04	33.75	43.30	34.78
M-P-P 13% NN	73232	19.26	36.07	51.73	34.11	70.99	34.64
Strip	106459.57	0.23	52.87			0.23	52.87
M-P-P 25% Strip	79819.62	0.23	52.87	296.03	34.66	296.26	34.67
M-P-P 20% Strip	85159.73	0.23	52.87	61.91	34.02	62.14	34.09

Table 6.4: Time is measured in seconds. The value of the Random Tour in the first line is the value implied by the pvar of randomly generated cities. All heuristics start from this tour. The abbreviations should be interpreted as follows: 1-P-P = One Pass Parallel tweaking, M-P-P = Multiple Pass Parallel Tweaking, and NN = Nearest Neighbor. Percent Figures indicate the tweaking was run until the specified percent improvement was achieved over the initial tour.

16K Cities			
Tour Type	Length	Time	
		Total	CM%
Random	1.3980667e8		
Strip	2368123	0.28	54.80
Equipopulated Strips	2369464	0.24	60.47

Table 6.5: Time is measured in seconds. The value of the Random Tour in the first line is the value implied by the pvar of randomly generated cities.

64K Cities Strip Tour			
Random Length	After Strips	Time	
		Total	CM%
2.2420385e9	1.8902352e7	1.00	89.05
2.237172e9	1.8913808e7	0.99	89.42

Table 6.6: Time is measured in seconds. The value of the Random Tour in the first column is the value implied by the pvar of randomly generated cities.

## Chapter 7

# Pseudorandom Permuter Chip

In this chapter we describe the design of a pseudorandom permuter chip. The chip routes 16 bit-serial messages from input pins to output pins to realize a permutation (i. e. a one-to-one correspondence between the input and the output ports). The output pin destination of each input can be determined externally, thus forcing a desired permutation, or it can be determined pseudorandomly. The chip contains a  $16 \times 16$  Benes network which is a permutation network based on the butterfly and composed entirely of primitive two input—two output switches. The settings of the control bits of these switches determine the permutation realized. The chip also contains a built-in pseudorandom number generator based upon a recurrence of polynomials over  $GF(2^8)$ . The chip was fabricated through MOSIS using  $4\mu m$  CMOS with one layer metal. Much of the chip design was inspired by Charles Leiserson.

The chip has the following features:

- Computation of the next pseudorandom switch settings is done while the current message is being sent. This allows immediate response to a request for a new permutation. While response is immediate, the message sent should be at least 25 bits long for this 16 bit version to insure theoretically good behavior.
- maximal period for pseudorandom number generator if properly initialized.
- built-in safeguards to help assure the pseudorandom number generator does not fall into a *trap state* where it will continuously generate nothing but zeros.
- buffers on internal broadcast control signals.
- sizing of vdd and gnd buses to avoid problems with electromigration.
- use of David Wise's highly regular interconnection strategy for butterfly-like networks.
- integration of logical function with architecture and physical layout
  - use of switch matrix as a computation matrix for the pseudorandom number generator.
  - interlacing of switch columns to make the logical values needed for the pseudorandom number generator physically close to each other.

- ability to explicitly set switches thereby forcing a permutation. Thus, the chip can form the basis of a permutation/routing network.
- ability to externally initialize all aspects of the pseudorandom number generator. This means the same chip can produce different pseudorandom permutation sequences, even when otherwise run identically.
- software support for generation of external control for operation and testing.

The remainder of this chapter is organized as follows. In section 7.1, we outline why one might want to build a pseudorandom permuter chip. We then provide the technical background used to make design decisions. Section 7.2 lists the network properties we require and describes the *Benes* network, which is the network we used. Section 7.3 describes pseudorandom number generation in general and describes the technique that we used. Section 7.4 gives a high-level overview of the chip architecture. Section 7.5 describes in more detail the response of the chip to external control signals. Section 7.6 describes the design and layout of interesting functional units in more detail.

## 7.1 Definition and Purpose

A pseudorandom permuter can be viewed as a box with  $n$  inputs and  $n$  outputs. When  $n$  bit-serial messages are placed at the inputs and the box is appropriately hit with control signals, the messages travel through the box. Each input is routed to exactly one output and each output gets exactly one input. That is, the outputs are a permutation of the inputs. Which line the input appears on is determined pseudorandomly by the permuter.

A pseudorandom permuter has two possible uses—both pertaining to routing of messages in a network. The first possible application is as an arbitrator. If there are, say,  $n$  wires coming into a network node, but only  $k < n$  going out, then in the case where there are more than  $k$  messages trying to go through the node, some number of messages must be dropped. At this point, each routing strategy must choose which messages to send through. In the absence of priority information, one could argue that the fairest method is to randomly choose winners. Just put the messages into a pseudorandom permuter, run it through a concentrator, and choose the leftmost  $k$  messages.

The second possible application is as a global means of congestion control in a network. Instead of sending messages immediately to their destinations, first send them “randomly” to some intermediate point, not necessarily on the path to their final destinations. Then route to final destinations. Valiant and Brebner [120,121] originally proposed this technique and proved it theoretically efficient. If the randomization is good, this process should smooth out local congestion. The network will never have to deal with worst case routing, but instead always handles the average case—the average, in fact, of our choosing. This scheme lengthens the base delivery cycle of the network, but now the networking software can be optimized for a known probability distribution of messages.

As we see later, we go to a lot of trouble to incorporate a pseudorandom number generator into the chip design. One could argue that it is easier to simply use a truly random source such as reading noise. We choose a pseudorandom generator because we want the chip to be testable

and we want it to be programable to a known state for testing the networks of which it is a piece.

## 7.2 The Permutation Network

This section describes some of the philosophy behind the design of the chip and gives details of the Benes network, the specific permutation network used on the chip.

We described the primitive switch in section 1.2. If we pseudorandomly set the control bit of a primitive switch, we have a pseudorandom permuter on two inputs. The chip design is a scaled-up version of this basic idea. We start with a network with  $n$  inputs and  $n$  outputs composed entirely of primitive switches. The network is capable of realizing any and all of the permutations on its  $n$  inputs without collisions and the permutation realized is determined entirely by the settings of the control bits of the individual switches. Then, by pseudorandomly setting these control bits, the network should act as a pseudorandom permuter on  $n$  inputs. Each resulting permutation should appear to be chosen randomly from the set of all permutations, providing the network does not overly favor some subset of permutations.

The Benes network described in section 1.2 meets all of the above stipulations. It is composed entirely of primitive switches, can realize any permutation of its inputs without collisions, and the resulting permutation depends only upon the control bits of the basic switches.

The only property listed above that does not follow immediately from the definition of the Benes network, is its capability of realizing any permutation on the  $n$  inputs without collision. For those who are familiar with graph theory, there is a simple proof of this fact based on matching in a bipartite graph. I will instead describe a simple algorithmic proof. Refer to figure 1-2 of the Benes network in section 1.2 to understand the labels used in the following tracing argument.

Say there is some given permutation you wish to realize. Here is how to set the switches to realize that permutation: Start at the top leftmost switch and set it arbitrarily. Since the setting does not matter let us choose 0. That means the first input  $in_0$  is routed through  $P_A$ , the upper subpermutation network, and the second input  $in_1$  is routed through  $P_B$ , the lower subpermutation network. Having just expended a degree of freedom, some switch settings are now forced. Find the output destination of  $in_0$ , say  $out_i$ . The output signal  $out_i$  emerges from a primitive switch,  $s$ , which has one input from network  $P_A$  and one input from network  $P_B$ . Signal  $in_0$  must travel through network  $P_A$ , so set the switch  $s$  such that the input from  $P_A$  is routed to  $out_i$ . This in turn forces the other output from switch  $s$ , call it  $out_j$ , to receive its signal from network  $P_B$ . That means the input to be routed to  $out_j$  must travel through  $P_B$ , which forces a switch setting in the first column, and so on. Continue the process, until you cycle back to the switch which was set arbitrarily to begin the process. In tracing back and forth, switch settings in the last (rightmost) column are forced by signals traced forward through network  $P_A$  and signals in the first (leftmost) column are forced by signals traced backward through network  $P_B$ . When we started, we sent input  $in_0$  through subpermutation  $P_A$ . Because we are routing a permutation, there can be no other messages originating at  $in_0$ . Thus when we cycle back we must be routing the message for  $in_1$ , and since this message is routed through subpermutation  $P_B$ , there is never a disagreement as to how to set the switch. If there are still unset switches in the outer columns, pick one of them, set it arbitrarily and repeat the above process.

The setting of the outer switches forces  $n/2$  signals into each of the two subpermutation networks. The desired permutation and the settings of the switches in the outer columns force the destinations of these signals within the subnetworks. By construction, no two signals routed through subnetwork  $P_A$  go to the same switch in the last column (similarly for subnetwork  $P_B$ ). Therefore, each of the subnetworks must realize a permutation of its  $n/2$  inputs. Recursively set the switches in the subnetworks to realize the forced subpermutations. When you get down to the last level, a single switch, the setting is forced. This is the algorithm used in the software I wrote to support testing of the chip.

Waksman uses the fact that we have that initial degree of freedom to put in an optimization. Rather than always starting with some switch and setting it to an arbitrary value, just hardwire it. In other words, eliminate one switch from one of the outer columns. This will eliminate a switch in each recursive invocation of the definition. My chip does not incorporate the optimization in favor of regularity and simplicity of design and layout.

In the Benes network and others like it, the settings of the switches determine the resulting permutation. It is not the case, however, that there is exactly one way to set the switches to realize a given permutation in a Benes network, or, for that matter, in any permutation network built of primitive switches. There are  $n!$  permutations on  $n$  inputs. If there are  $r$  switches in the network, however, then there are  $2^r$  ways to set all switches simultaneously. For  $n > 2$ , there is no  $r$  such that  $2^r = n!$ . To be able to achieve any permutation, we must choose  $r$  such that  $2^r > n!$  or in other words,  $r > n \lg n$ . This means that some distinct settings of the switches will result in the same permutation. If some subset of the total permutations appears most of the time, then our strategy of pseudorandomly setting switches will not result in a good pseudorandom permuter. The Benes network seems to have sets of permutations which are equally favored with some sets more favored than others. The basic idea is that whenever the two inputs that enter a primitive switch in the first row are destined to two outputs emerging from the same primitive switch in the last column, then we have a degree of freedom in realizing the permutation. In this thesis, we have omitted a detailed description of our preliminary analysis.

### 7.3 Pseudorandomly Setting Bits

This section describes the issues involved in choosing a pseudorandom bit generator: what we would like, why we cannot have it, what we ended up with, the pros and cons of the final choice, and how the algorithm translates into hardware operations.

Theoretically the optimal way to determine the settings for the control bits of the switches in the network would be to use a *cryptographically strong pseudorandom bit generator* (CSPRBG). A CSPRBG is an algorithm for generating pseudorandom bits based upon some intractability assumption. The algorithm requires a function that is easy to compute (polytime), but hard to invert (no known polytime algorithm for inversion). If the assumption holds (i. e. inverting is hard), then there is no polynomial time algorithm that can distinguish between the output of the generator and truly random bits [18,50,52,80,109].

This is certainly all one could hope for in a pseudorandom bit generator. Unfortunately, the known CSPRBG algorithms have some problems, particularly for this application. The simplest CSPRBG is based upon the assumption that it is hard to factor products of two primes where the primes are roughly equal in size. The best algorithm for generating cryptographically

strong pseudorandom bits using this function requires a multiplication of two  $k$ -bit numbers mod another  $k$ -bit number to get  $\lg k$  bits out [3]. This is very expensive in terms of hardware and clock cycles, especially since  $k$  must be large before any of the theory mentioned above applies. The multiplier and modulo circuits would occupy the whole chip. Each multiplication and modulo operation could take many cycles and/or slow the clock down. Also because each operation only produces  $\lg k$  bits, we would have to perform a lot of them to generate all the bits we need for the network switches.

In addition to the great expense, the cryptographically strong system does not integrate well with the Benes network. All area between columns is needed for routing between switches, so whatever pseudorandom bit generator is on the chip has to run within columns or sit above the network. The  $\lg k$  bits would have to be produced in a central location and shipped to the switches which would cause nasty control problems, long buses, and perhaps wasted space.

Forced to sacrifice "randomness" for speed, simplicity, and feasibility, we decided to use a linear shift register scheme. Linear shift registers produce pseudorandom numbers, not pseudorandom bits which is what we want. However, there is a certain amount of theory to support them. The discussion to follow contains, in the interest of conciseness and relevance, largely definitions and mechanics. Refer, for example, to [7] for more details on the theoretical aspects of this type of pseudorandom number generator.

The key to understanding the theoretical merit of the algorithm used on the chip is a shift in perspective. The arithmetic systems we grew up with, such as addition and multiplication over the natural numbers or the real numbers, are not the only systems that behave "normally". Appropriate definitions of elements, and the binary operations of "addition" (+) and "multiplication" (·) operating over the elements result in an arithmetic system known as a *field*. Fields obey many of the laws of standard arithmetic such as the distributive law, existence of identity elements for addition and multiplication, etc. For more information on fields and to get some insight into theoretical discussions of shift register schemes, refer to an introductory algebra text such as [21,81].

One of the simplest fields is called  $GF(2)$ , short for Galois field of two elements. The elements are  $\{0, 1\}$ . Addition (+) and multiplication (·) are standard addition and multiplication respectively mod 2. That is, add/multiply two elements, then divide by 2 and take the remainder. With these definitions, addition and subtraction are both equivalent to the logical exclusive or of two bits. I will abbreviate exclusive or as xor or the symbol  $\oplus$ . Multiplication becomes logical and ( $\wedge$ ).

We can now take our new arithmetic system one step further and use field elements as coefficients of polynomials. Addition and multiplication remain the definitions we are used to except that addition/multiplication of coefficients is performed with the addition and multiplication operators of the field. The case of the field  $GF(2)$  is particularly useful. Now  $n$ -bit sequences can represent polynomials of degree  $n - 1$  with coefficients over  $GF(2)$ . These polynomials are used so commonly, they have a special name:  $GF(2^n)$ . For example  $10010 = x^4 + x$  is an element of  $GF(2^5)$ .

If we are generating  $n$ -bit pseudorandom numbers, a simple linear recurrence relation over  $GF(2^n)$  will give *maximal period*. All linear schemes will cycle at some point. Those with maximal period generate  $2^n - 1$   $n$ -bit values before cycling. The value missing is 0. This is a *trap state* from which only 0 can be generated.

The recurrence relation used to generate pseudorandom numbers on the chip is:

$$y_i = (x \cdot y_{i-1} + y_j) \bmod P$$

The  $n$ -bit values  $y_i$ ,  $y_{i-1}$ ,  $y_j$ , and  $P$  are all elements of  $GF(2^n)$ . The bit representation of polynomial  $y_i$  is the next pseudorandom number generated. The bit representation of polynomial  $y_{i-1}$  is the last pseudorandom number generated and  $y_j$  is one generated in the past preferable not too close to  $y_{i-1}$ . Assuming that some fixed number of values are stored, we use the one generated longest ago. The variable  $x$  is the variable of the polynomial. The polynomial  $P$  is a *primitive polynomial* over  $GF(2^n)$ . For our purposes, the only important properties of primitive polynomials are that the leading coefficient (the coefficient of  $x^{n-1}$ ) is 1 and that for any given  $n$  there are published tables of primitive polynomials for  $GF(2^n)$ . I used the small one in the appendix of [7] for testing. Since the leading coefficient of a primitive polynomial is always 1, we can represent a primitive polynomial of  $GF(2^n)$  using only the least significant  $n - 1$  bits.

Although the recurrence appears nasty, it translates into a very simple algorithm with very simple hardware requirements. The first thing to notice is that  $P_1 \bmod P$  where  $P_1$  is a polynomial over  $GF(2^n)$  and  $P$  is a primitive polynomial over  $GF(2^n)$  is quite simple to compute. If the leading coefficient of  $P_1$  is 0, the quotient  $P_1/P = 0$  and the remainder is  $P_1$ . That is, the operation has no effect because  $P > P_1$ . If the leading coefficient of  $P_1$  is 1, then the quotient  $P_1/P = 1$  and the remainder is

$$\begin{aligned} P_1 - P &\approx P_1 + P \\ &\approx P_1 \oplus P. \end{aligned}$$

Thus, the result of  $P_1 \bmod P$  will always have leading coefficient 0.

The mod operator distributes so the recurrence relation can be simplified:

$$y_i = (x \cdot y_{i-1}) \bmod P + y_j \bmod P$$

Because polynomial  $y_j$  was generated earlier via the recurrence relation, it must have leading coefficient 0 so  $y_j \bmod P = y_j$  and

$$y_i = (x \cdot y_{i-1}) \bmod P + y_j$$

There are  $r = n/2$  rows of switches in a Benes network with  $n$  inputs and  $c = 2 \lg n - 1$  columns. The control bits of the switches in a row form a  $c$ -bit number, or alternatively an element of  $GF(2^c)$ . We will, in fact, look at it as an element of  $GF(2^{c+1})$  with leading coefficient 0 because each polynomial will be produced by a recurrence relation of the stated form with the final modulo operation involving a primitive polynomial over  $GF(2^{c+1})$ . The switch matrix has  $r$  such numbers, so in the recurrence,  $y_j$  might be better subscripted  $y_{i-r}$ .

The multiplication by  $x$  makes the coefficient of  $x$  be the coefficient of  $x^2$ , the coefficient of  $x^2$  be the coefficient of  $x^3$ , etc. For example  $(x^5 + x^4 + x^2)x = x^6 + x^5 + x^3$  as expected. Looked at in the bit representation, it is just a shift left with the right side filled with a 0.

To generate a new  $c$ -bit pseudorandom number, the chip takes the last value generated and shifts it left. If the bit shifted out is a 1 (that is,  $x \cdot y_{i-1}$  has leading coefficient 1), then it xor's the shifted value with a primitive polynomial over  $GF(2^{c+1})$ . If the bit shifted out is a

0, it does not do the xor. Finally, it xor's this intermediate value with the  $r$ th oldest value calculated. This corresponds to adding in  $y_{i-r}$ .

This algorithm has some good points. Firstly, it meshes very well with the architecture of the Benes network. The switch matrix can serve as a matrix of values calculated in the recurrence. If the switches are interconnected within the columns, calculated values shift along when new pseudorandom numbers are generated. After  $r$  repetitions of the algorithm, all switches have new settings. Secondly, the control and hardware necessary to implement this scheme are simple. The hardware consists of xor gates and registers. It is also compact enough to fit into the padframe with a  $16 \times 16$  Benes network, and it is fast enough to run in realtime.

There are also some major disadvantages in terms of performance as a pseudorandom bit generator. First of all, not all switch settings are possible from the generator since we can never get even two rows with identical settings, not to mention three rows, etc. It is possible, then that the chip cannot be set pseudorandomly to realize all permutations—a serious shortcoming. Then, of course, there is the problem of whether maximal period is really a sufficient definition of randomness, dismissing for the moment the concerns of space and time that forced the decision. After all, it is periodic. The way the chip actually runs will cause some configurations to be skipped in any given cycle. This makes the period even shorter, though it might make behavior less consistent.

## 7.4 Architecture

This section gives a quick view of the overall architecture of the chip highlighting the main architectural features. It begins the discussion of timing as it effects each unit, and gives those details of the functional units' substructure needed to understand the next section on operations. For more details of the structures of each unit, see section 7.6.

The chip has five main functional units: a control unit, a polynomial register, a main register, a switch network, and a set of input latches. Figure 7-1 illustrates the communication between units. The chip runs on a nonoverlapping  $\phi_1, \phi_2$  clock. Throughout this discussion "on  $\phi_1$ " means the same thing as "when  $\phi_1$  is (or goes, depending upon context) high (logic 1)". The same is true of "on  $\phi_2$ ".

The control unit accepts control information from off-chip, specifically a two bit op code and a reset signal, and produces control signals for the on-chip circuitry. It also sends a signal called **flag** directly to a pin to indicate the completion of some operations. The control unit is implemented as the standard one-cycle PLA from HPLA and has the timing described in the HPLA manual. Inputs are latched on  $\phi_1$  and outputs are available, with potential glitches, on the next  $\phi_2$ . The glitches do not present problems because the signals are not actually used until at least the following  $\phi_1$ .

The polynomial register holds the least significant 7 bits of the primitive polynomial used in generating pseudorandom numbers. As mentioned in section 7.3, the leading bit must be 1, so there is no need to store it. On  $\phi_1$ , based upon the value of a control signal, the register either loads a new value from off-chip or holds its current value. The new value is available immediately on the polynomial register's outputs. A piece of combinational logic generates subcontrol signals which pass through superbuffers and are broadcast to 7 load/hold register cells. From now on, we refer to this register as the poly register for short.

The main register serves as an accumulator during the generation of each pseudorandom

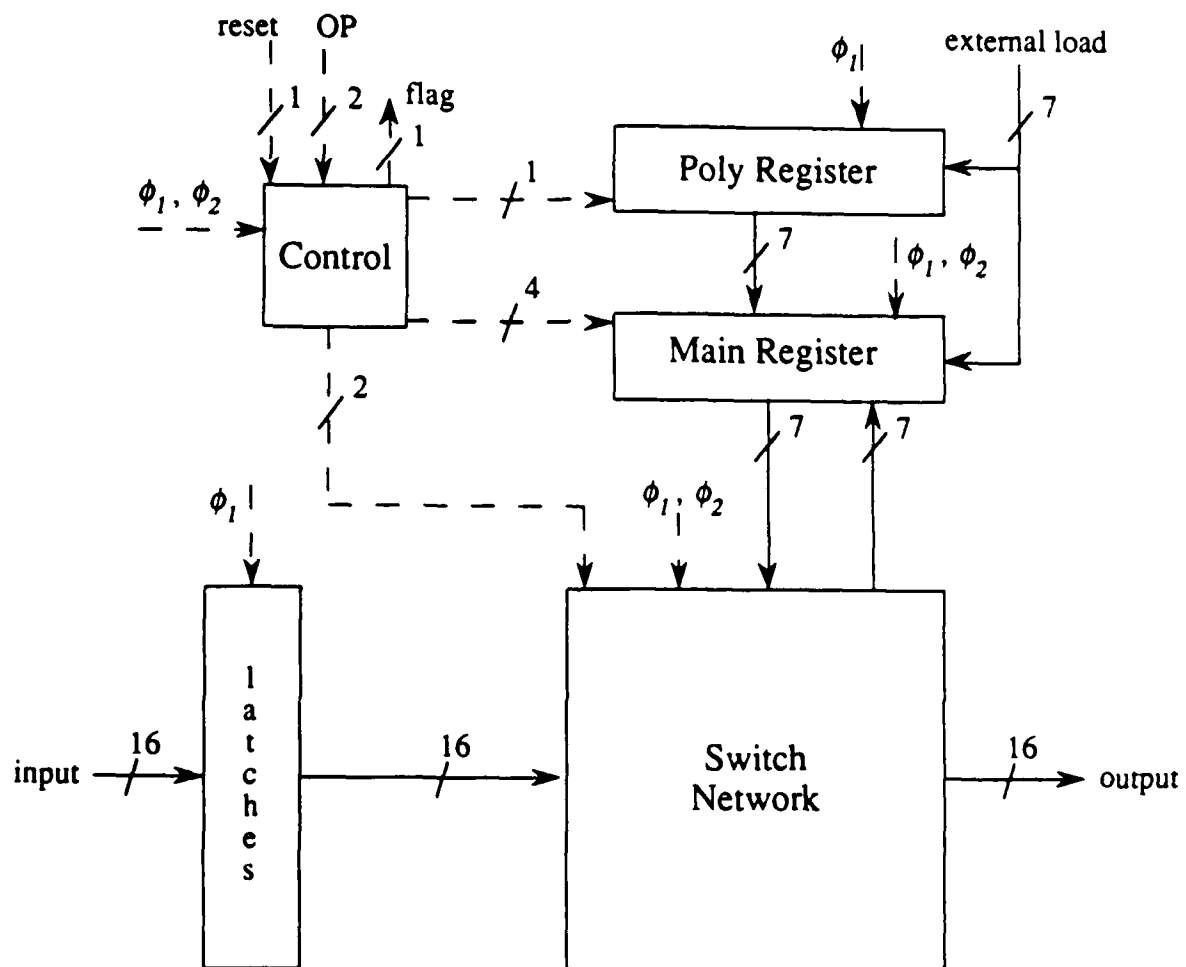


Figure 7-1: The architecture of the chip. Dashed lines indicate control signals. Solid lines indicate data signals.

number. It holds a 7 bit number, conceptually a member of  $GF(2^8)$  with leading coefficient 0 at the end of each computation. Each  $\phi_1$ , the main register performs one of the following actions based upon the values of its four control inputs:

1. load a new value from off-chip,
2. perform a bitwise xor of its current value and the current value of the poly register and load the result,
3. perform a bitwise xor of its current value and the value available from the switch network and load the result,
4. hold its current value,
5. or shift left.

The value loaded on  $\phi_1$  is available at the main register outputs on the following  $\phi_2$ . As with the poly register, a separate piece of combinational logic generates subcontrol signals which pass through superbuffers and are broadcast to 7 register cells.

The switch network is the heart of the chip. It routes the 16 messages to the 16 output ports. The destinations of the messages are determined by the values of the individual switch controls. This and many of the other properties of the Benes network were described in sections 7.2 and 1.2. For a 16 input Benes network there are 56 switches: 8 rows of 7 switches each.

The switch is a bit more complicated than the simple box in figure 1-1 would indicate. Figure 7-2 shows a more detailed view of the switch with all its support circuitry. There are three major subpieces of a switch: the **next-set** register, the **current-set** register, and the front end.

The **next-set** register holds the value the control would be set to if the chip were to go into a new configuration to realize a new permutation. The value of this register changes during calculation of new pseudorandom numbers. On  $\phi_1$  the register either loads a new value or keeps its old value depending upon the value of its control signals. The value loaded is available at the register output on the following  $\phi_2$ .

The **current-set** register holds the value of the switch control. This value holds steady whenever messages are being sent. On  $\phi_2$ , the register either loads a new value from the output of the **next-set** register or holds its current value depending upon the values of its control signals. The new value is available immediately. Figure 7-2 shows four control signals for this register and for the **next-set** register. These are just clocked load/hold signals and their inverses. All four come from a single signal sent by the control unit.

The front end is essentially the box shown in figure 1-1. Because the circuitry within the box is implemented with pass transistors, each message signal also passes through an inverter. The gates of the inverters serve as a site for charge storage if there were a reason to latch at one or more places in the network. As it turns out, the switch network through which the message bits flow is purely combinational. The inverters serve only in their secondary capacity as boosters for the signals.

Each column of the switch network has its **next-set** registers connected into a big shift register. They share control lines so they all either load or hold simultaneously. Each column has two connections to the main register at either end of the shift path. Whenever the main

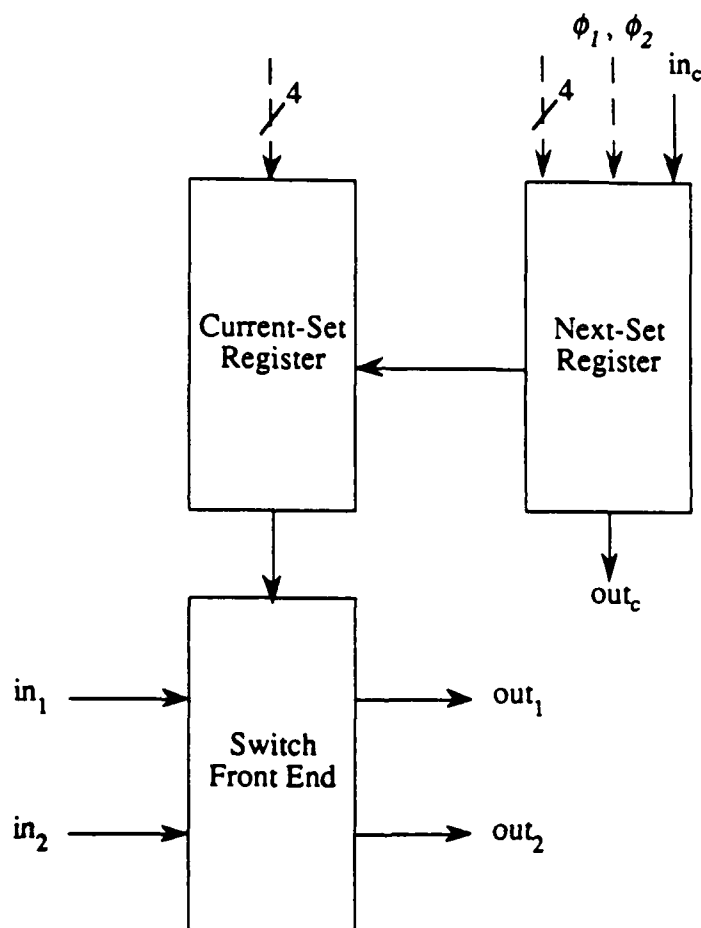


Figure 7-2: Subdivisions of a switch. Dashed lines indicate control signals. Solid lines indicate data signals.

register holds a newly computed pseudorandom number, the **next-set** registers load. This causes the newly calculated value to flow into the **next-set** registers of the top row of switches. Meanwhile all the old **next-set** registers values shift along the path. The row that contains the last register on the shift path holds the pseudorandom number calculated longest ago. This is the value added in at the end of the calculation of a new value, so it participates in the second connection to the main register. Because the two ends of the shift path should be physically close to each other for easy connection to the main register, each column is *interlaced* as shown in figure 7-3. As mentioned in the discussion of the **next-set** register, the values load on  $\phi_1$  but are not available on outputs till the following  $\phi_2$ . Thus whenever a new pseudorandom number is loaded into the first row, all the old values travel exactly to the next row as specified in the interlace pattern. Signals can never travel two arcs at once.

Control signals for the switches are all bused vertically within the columns since all the space between columns is needed for the butterfly interconnection pattern. A separate piece of combinational logic generates the control signals which then pass through superbuffers and are broadcast to the base of each column. Each column then has another set of superbuffers to power the signals through the columns.

The last functional unit, the input latches, is also the simplest. All 16 message inputs are latched on  $\phi_1$ , so the input signals can change asynchronously. This gives a one bit per clock cycle throughput on the messages. Signals emerge from the latches inverted. This results in uninverted signals at the output ports because there are an odd number (7) of switches through which each message bit passes and each switch inverts its inputs.

## 7.5 Operations

The chip has four operations:

- load the poly register
- load a value into the top row of switches (shifting old values down)
- send messages
- get a new permutation.

This section describes each of the four operations in detail. It explains what each functional unit does at each time step of each operation, and, where necessary, explains why certain steps are performed.

### 7.5.1 External Control

There are four signals used to control the chip externally: *reset*, a two-bit opcode, and *flag*. As far as external control is concerned, the point at which  $\phi_1$  rises is the only time of importance. That is when the input control signals (*reset*, *op code*) are noticed and it is when *flag* is valid.

Whenever the *flag* goes high, it means the chip has completed whatever it was doing and will look at and perform the operation specified by the *op code* on the *next*  $\phi_1$ . The *flag* signal is not valid until  $\phi_1$ , so there is a whole clock period in which to set the *op code* and data lines.

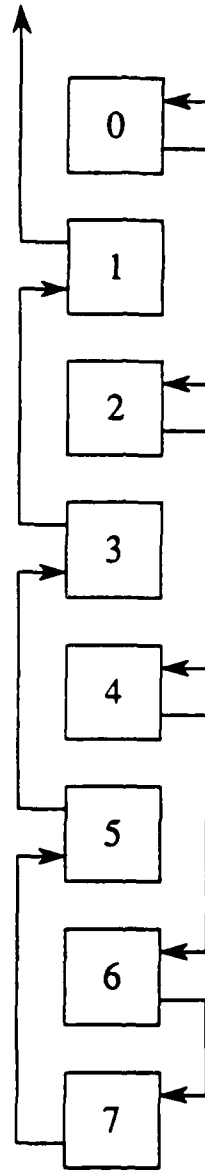


Figure 7-3: Interlacing within the columns allows easy connection to the main register. The storage location of the newest and oldest values are now physically close. When the **next-set** bits shift, each bit travels along the arrow originating in its current row.

If the reset signal is high on  $\phi_1$ , the chip will stop whatever it was doing and raise the flag indicating that it is ready to respond to an op code on the next  $\phi_1$ . This signal helps initialize the chip by forcing the control unit into a known state. A complete initialization of the chip includes setting the poly register and all switches (i. e. performing the load switches operation 8 times). If the reset signal is not lowered, the flag will remain high and the chip will do nothing. The reset signal should not be used when one cares about any data currently in the chip's registers. Do not make any assumptions about internal values immediately after a reset.

For the operations of loading the poly register, loading a switch row, and getting a new permutation, the external control sequence is as follows:

1. When flag goes up, set the op code and any external data signals involved in the operation. The flag will go down after being up for exactly one clock cycle (as long as reset is low).
2. Clock the circuit until flag rises again.

All external signals are latched so they can change as long as they are stable during the  $\phi_1$  in which the op code is read (i. e. one cycle after the flag rises). Even the op code can change after it is read.

The operation of sending messages is controlled differently:

1. When flag rises, set the opcode and the message inputs.
2. Clock the circuit. The chip will send through one message bit per cycle. Hold the bits stable through  $\phi_1$  and change to the next values of the message bits during  $\phi_2$ . The flag will not rise.
3. Whenever an op code other than the one for sending messages is placed on the op code pins, the chip will react to it on the first possible  $\phi_1$ . While in the other operations the op code is ignored after start up and therefore can be changed. For this case, the op code *must* be stable for the entire message sending process.
4. After a new op code is placed on the pins, continue clocking until the flag rises to signal the completion of the new operation.

The remainder of this section will give details of each operation starting at the  $\phi_1$  immediately following the one where the op code is noticed.

### 7.5.2 Load Poly Register (op code 00)

In the first cycle of this operator, the poly register loads from the external data lines. All other registers hold. On the second and last cycle, the main register xors its current value with the value just loaded into the poly register and loads the result. The flag rises at the completion of this cycle.

The latter action is a safety measure for the pseudorandom number generator. If the value in the main register is 0 and the value of all the *next-set* registers is 0, then the generator enters a trap state from which it generates only 0. That is, the recurrence becomes  $y_i = 0 \cdot x + 0$ . Although this situation seems artificial, it can happen quite easily. If for some reason one

wanted to force the identity permutation via external switch setting, the easiest way to do it is by setting all switches to 0. In fact, the software support I wrote to test the design would do it that way. The switch setting operation leaves the **next-set** registers with the same values as the **current-set** registers.

The xor should ensure that the pseudorandom number generator will not start off in a trap state. In fact, it does not as long as the polynomial loaded in is indeed the lower 7 bits of a primitive polynomial representation. Primitive polynomials *must* have lowest bit 1. That is another property of primitive polynomials not mentioned in section 7.3. Thus if the main register reads all 0's, it will be changed to have at least one 1. If by some bad luck the xor actually *made* the value in the main reg be 0, then we are safe anyway. For the xor to yield 0, the original value in the main register, which is the same as the value of the **next-set** registers in the first row, must have been nonzero.

### 7.5.3 Load a Switch Row (op code 01)

In the first cycle of this operation, the main register loads a value from the external data lines. These are the same lines the poly register loads from in op code 00. The value is read on  $\phi_1$  and available on the main register outputs on  $\phi_2$ . All other registers hold.

The second cycle is more complicated. On  $\phi_1$ , the value in the main register is xor'd with the value in the poly register and loaded back into the main register. This new value will not show on the main register outputs until  $\phi_2$ . This action is performed for the same reason it was done on the load poly register operation: to prevent a trap state in the pseudorandom number generator. Since both the load poly register and load switches operations do this, the order of initialization does not matter.

Meanwhile, during the same  $\phi_1$  as the xor, the **next-set** registers are loading. The value in the main register (i. e. the value loaded from the external signals) is loaded into the first row and all other values shift along in accordance with the interlace pattern. Thus performing this operation 8 times will set all switches. On  $\phi_2$ , the new values in the **next-set** registers become valid and the **current-set** registers load. The poly register holds through the whole operation. The flag signal rises at this point to be noticed on the next  $\phi_1$ .

### 7.5.4 New Permutation (op code 10)

This operation is very simple. The **current-set** register on each switch loads from the **next-set** register on  $\phi_2$ . All other registers hold. The flag signal goes up to be noticed on the next  $\phi_1$ .

### 7.5.5 Send Messages (op code 11)

The actual act of sending the messages is very simple. On each  $\phi_1$ , the values in the message inputs are latches and propagate through the combinational network of switches. The switch **current-set** registers and the poly register hold steady whenever messages are sent.

To give quick response to a request for a new permutation, the pseudorandom number generator circuitry works in the background while the messages pass through. It operates as follows: on the first  $\phi_1$ , the main register shifts left while the **next-set** registers hold. If the bit that is shifted out is a 1, then on the following  $\phi_1$ , the value in the main register is xor'd with the value in the poly register and loaded back into the main register. Then on the next  $\phi_1$ , the

value in the main register is xor'd with the value in the second row of switches (oldest). If the value shifted out in the first step is 0, the chip skips the xor with the poly register and goes straight to the xor with the values at the end of the *next-set* register shift path. Following the latter xor, the *next-set* registers load while the main register shifts to start the whole process over again.

As long as the messages being sent are at least 25 bits long, all 8 rows will be replaced (that is,  $8 \times 3$  cycles +1 for lag on the first load. If the messages are longer or there are 0's shifted out in some of the computations, the values in all the rows will be new, but the computation will continue. There will just be skipped configurations.

External switch settings add extra "randomness" while the bits in the switches shift out and are used in the calculations. Afterwards, the pseudorandom number generator settles into a maximal period.

### 7.5.6 Switching Out of Message Sending Mode

As mentioned before, the way to stop sending messages and perform one of the other three operations is to change the op code. The chip stops pumping through messages. Its exact actions, however, depend upon the new op code. If the new op code is to load a switch row or load the poly register, the chip aborts the computation of the pseudorandom number generator. The new operation will scramble the main register contents. Also, these operations indicate a new initialization. If the new op code is to get a new permutation, the *current-set* registers load immediately. Then the chip finishes the last computation of the pseudorandom number generator before raising the flag. This is so that in a steady state use of ... new permutation, send messages, new permutation, send messages, ... the pseudorandom number generator operates as expected, not scrambling its computations in the middle.

## 7.6 Logic Design and Layout

This section gives details of the actual implementation of the architecture described in section 7.4. The first subsection describes the overall design and layout concerns and priorities. The second subsection gives details of the implementation of individual functional units layout methodology for any unusual units. The final subsection criticizes the implementation, indicating where it could be improved.

### 7.6.1 Overall concerns

Five major rules shaped the final logic design and layout of the chip. First of all, we designed with scaling in mind. We made design decisions as though the chip were to have 32 or even 64 message inputs rather than the 16 it was feasible to implement. Secondly, we sized power and ground buses to avoid problems with electromigration. Thirdly, we placed body plugs at least every  $50\lambda$  within active areas. Fourthly, we used nand gates rather than nor gates in random logic. The last rule almost goes without saying: we tried to make the layout as compact and regular as possible, avoiding layer changes, etc.

The scaling rule influenced the shape of most of the functional units. It was clear that the Benes network of switches and interconnect would take a fair amount of space. Because each

cell of the main register has to communicate bitwise to a column of the switch network and bitwise with cells of the poly register, it makes sense to place the main register above the switch network and the poly register above the main register. Theoretically, however, this would seem to be the dimension we could least afford to put these, especially if the design is to scale to larger networks. For a Benes network of  $n$  inputs, the number of columns grows as a function of  $\lg n$ , specifically  $2\lg n - 1$ , while the number of rows grows as a function of  $n$ , specifically  $n/2$ . As  $n$  grows, the number of rows goes up exponentially faster than the number of columns. In laying out the individual switches, therefore, we tried to squeeze down the vertical dimension (the one with  $O(n)$  growth in the network).

This decision in turn influenced the rest of the chip. The switch layout is indeed a very skinny rectangle. When  $n = 16$ , the asymptotic behavior of the network is not yet apparent. In fact, the numbers in rows and columns are almost equal. Although the switch layout may be perfect for a 32 or 64 input network, it results in a skewed 16 input network. The rows are much longer than the columns and, in fact, fill virtually the whole pad frame in that dimension.

Figure 7-4 shows the final floorplan, more or less to scale. We had been worried about finding room to place the main register, poly register, and superbuffers for the switch control signals above and below the switch network. We were forced to place *all* remaining circuitry in that dimension and found that we had plenty of room. In fact, to make interconnection between the switches and registers easier, We tried to squeeze the register control combinational logic horizontally, exactly the opposite of what we did for the switches.

Sizing of power and ground buses unfortunately did not figure into the early planning of the chip. It was not until the layout was almost completed that a colleague pointed out that the chip would die instantly of electromigration. The chip originally had all power and ground lines a uniform  $4\lambda$  wide. As it turned out, the fix was easy. The major power and ground buses (superbuses) that come off the pins and feed the switch network are  $50\lambda$  wide to match the width of the lines at the pins. This is much more than actually needed. The lines narrow down to  $4\lambda$  within cells except for within the pla. The pla was generated by hpla and we did not quibble with the sizes chosen by the original designer. The widest rails of the pla are attached to the superbuses by lines of the same width as the pla rails.

To avoid the problem of latch up we tried to adhere to the rule of body plugs every  $50\lambda$  within the guts of the chip. Wherever there are gates, PN butting contacts make the job simple. It is more difficult to do in areas where there is just interconnect or pass transistors.

The switches have places at either end where a vdd rail extends past the area where it is required for gates so it can power a body plug. In both cases, this extension does not add to the size of the cell. In places where the extra body plug would increase the basic cell, the optimization is ignored. After all, the latch up problem is not as major in areas where there are no gates and the  $50\lambda$  figure is just a rule of thumb.

There are three areas of the chip with body plug spikes. These are runs of vdd with nothing attached but a series of body plugs spaced approximately  $50\lambda$  apart. They are in areas of essentially open space between cells where the vdd line is running anyway. The three areas are between the cells of the poly register, between cells of the main register, and between superbuffers cells on the bottom. The spikes are spaced about every  $50\lambda$ .

The fourth rule of thumb influenced logic design: in CMOS, nand gates are better than nor gates because the pullups are in parallel, speeding up the slower P-type transistors. The chip uses a nor gate only as a subpiece of the xor gate. All other logic uses only 2-, 3-, and 4-input

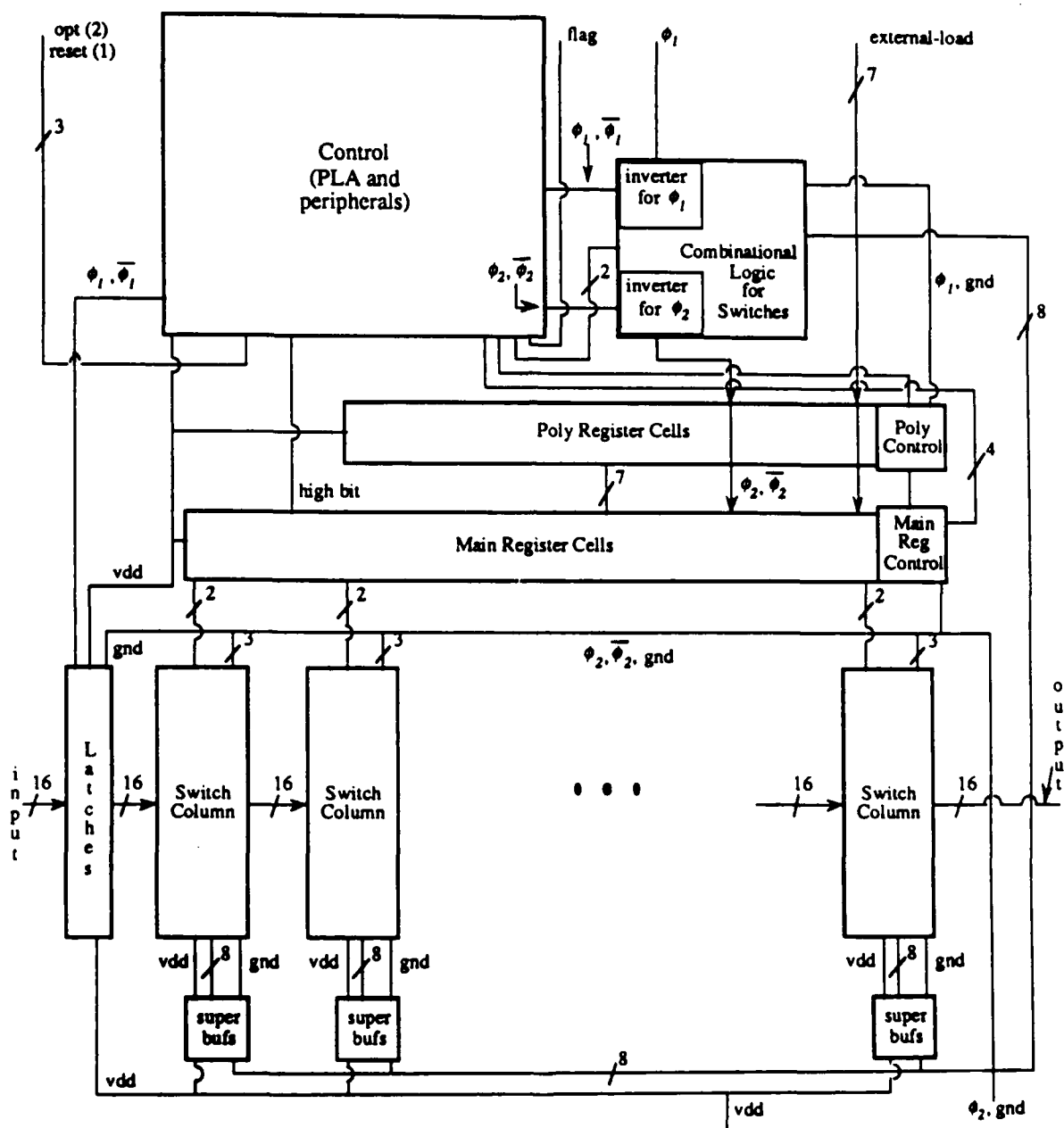


Figure 7-4: The floorplan of the chip.

nands, inverters, and pass gates. The choice of nands did not even add to the number of gates or transistors.

### 7.6.2 Implementation of Individual Units

#### The Control Unit

The control unit has three subparts. The first is a one-cycle pla generated by HPLA, specifically, CMOS4ONECYCLE. The second is a row of pass transistors gated by  $\phi_1$  which latch the pla inputs. The third is the peripheral circuitry on the outputs of the pla. A row of inverters guarantees a place for charge storage. The state bits used to determine the pla's next actions travel through another set of inverters and back to pla inputs. The flag control signal is also inverted once again. All other pla outputs participate in combinational logic elsewhere. Appropriate choices of gates in these combinational networks compensates for the signals' inversion. In fact, the inversion usually makes nand gates easier to use and is therefore a bonus.

These are the control signals generated by the pla. The "p", standing for pla, in front of the name distinguishes the signals emerging from the pla from those that arrive at final destinations after one or two inversions:

- **ppolyl**—The one poly register control line. If it is high, the poly register loads from external lines on the next  $\phi_1$ . Otherwise it holds.

The next four signals control the main register.

- **phold**—if high, the main register holds its value on  $\phi_1$ . Otherwise, the other signals determine the behavior.
- **pxor**—if **phold** is low and this is high, the main register performs an xor as specified by **pxornorth**.
- **pxornorth**—if **phold** is low and **pxor** is high, then this tells the main register how to xor: if it's high, xor with the value from the poly register, and otherwise, xor with the value from the switches. If **phold** is high or **pxor** is low, this signal is ignored.
- **pload**—if **phold** and **pxor** are low, then this signal specifies the operation of the main register: if high, load a value from the external lines, and otherwise, shift left.

The two switch control signals:

- **pnext**—if high, the **next-set** registers in the switches load on the next  $\phi_1$ . Otherwise, they hold.
- **pcur**—if high, the **current-set** registers in the switches load on the next  $\phi_2$ . Otherwise, they hold.

External control:

- **flag**—indicates completion of some operations. See the discussion in section 7.5.

The one-cycle pla generated by HPLA has the advantage of quick generation of signals and quick response to outside control. It also eliminates the need for an extra register to hold the value shifted out of the main register in pseudorandom number generator calculations. The one-cycle pla is ready to use that information before it disappears and produces the signals the necessary control signals to give the main register maximal throughput.

The one-cycle pla does have some problems though. The glitches that the HPLA manual warns about do not effect the chip because the chip is entirely synchronous and nothing uses a signal until it has had time to settle. The two real problems are solved with additional circuitry. First we must add latches for the inputs because asynchronous changes in the signals will effect the operation of the pla. The fact that many of the signals will change on  $\phi_2$  makes the latches imperative.

The second problem is that the outputs go through pass transistors, but there are no gates upon which to store the charge. Because there are many stray capacitances in VLSI chips, any gate that is supposed to store charge must be as close to the pass transistor as possible. This ensures that the gate capacitance far outweighs the stray capacitances. In order to do this, we yanked out a row of inverters from within the pla just above the pass transistors. These inverters are very dense and roughly aligned with the outputs. We then adjusted the inverters until they fit onto the outputs without violating design rules.

### The Switch Network

The only interesting thing about the chip control circuitry is that it happens to be in a convenient place to invert the clocks. The clock inverters are 9 : 1 on the assumption that the true signals are driven as hard as necessary externally. The pass transistor network was fairly difficult to layout, with four pairs of interconnected transistors gated by two control signals. Nevertheless, the layout did turn out to be fairly compact.

Figure 7-5 illustrates how the interlacing is actually implemented. There are two data paths for the next control bit: one flowing down, the other flowing up. The even switches (0, 2, 4, etc. numbered from the top) take input from and place output on the downward line only. The odd switches use only the upward line. The end switches (a special case of an even switch) take input from the downward line and place output on the upward line. For regular even and odd switches, whichever line is not being used passes through uninterrupted while the line that is used is interrupted by the **next-set** register of the switch. For the odd switches, the "uninterrupted" line has to make some detours in the actual layout. All types of switches—even, odd, or end—are pitch aligned to share all control signals, both **next-set** data paths, power, and ground.

Figure 7-6 is a copy of David Wise's proposal for a butterfly interconnection layout [126]. This layout has exactly the same functionality and properties of the regular butterfly layout. It even has the same recursive divisions. The interconnection between columns 0 and 1 involves all 8 switches. The interconnection between columns 1 and 2 consists of two copies of a smaller pattern involving 4 switches, and so on.

The Wise interconnection pattern has some advantages over the traditional interconnect pattern. First of all, all the lines are roughly the same length (exactly if we had 45 degree lines). Secondly, it is totally symmetric so that instead of reflecting interconnect pieces to form a Benes network, either a reflection or a rotation will do. More importantly, for layout, only a translation is needed. Also it is clear how to lay it out in VLSI (with 45 degree lines allowed): route northwest traveling lines on one layer, say poly, and northeast traveling lines in the other

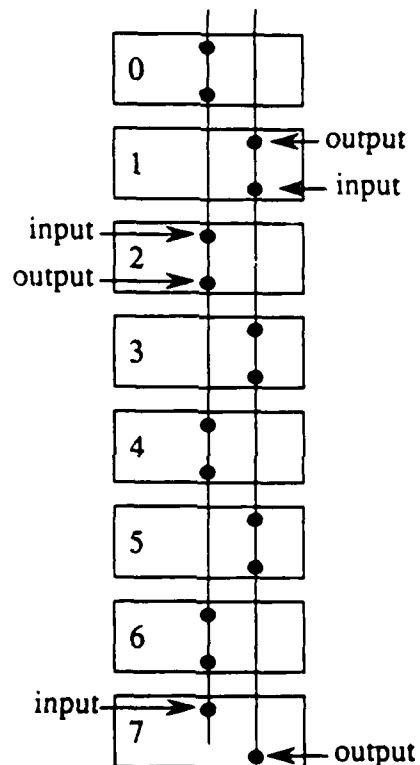


Figure 7-5: The implementation of the interlacing. There are two data lines for the **next-set** registers. The one on the left sends data down. The one on the right sends data up. Connection points for each switch depend upon the switch's location within the column.

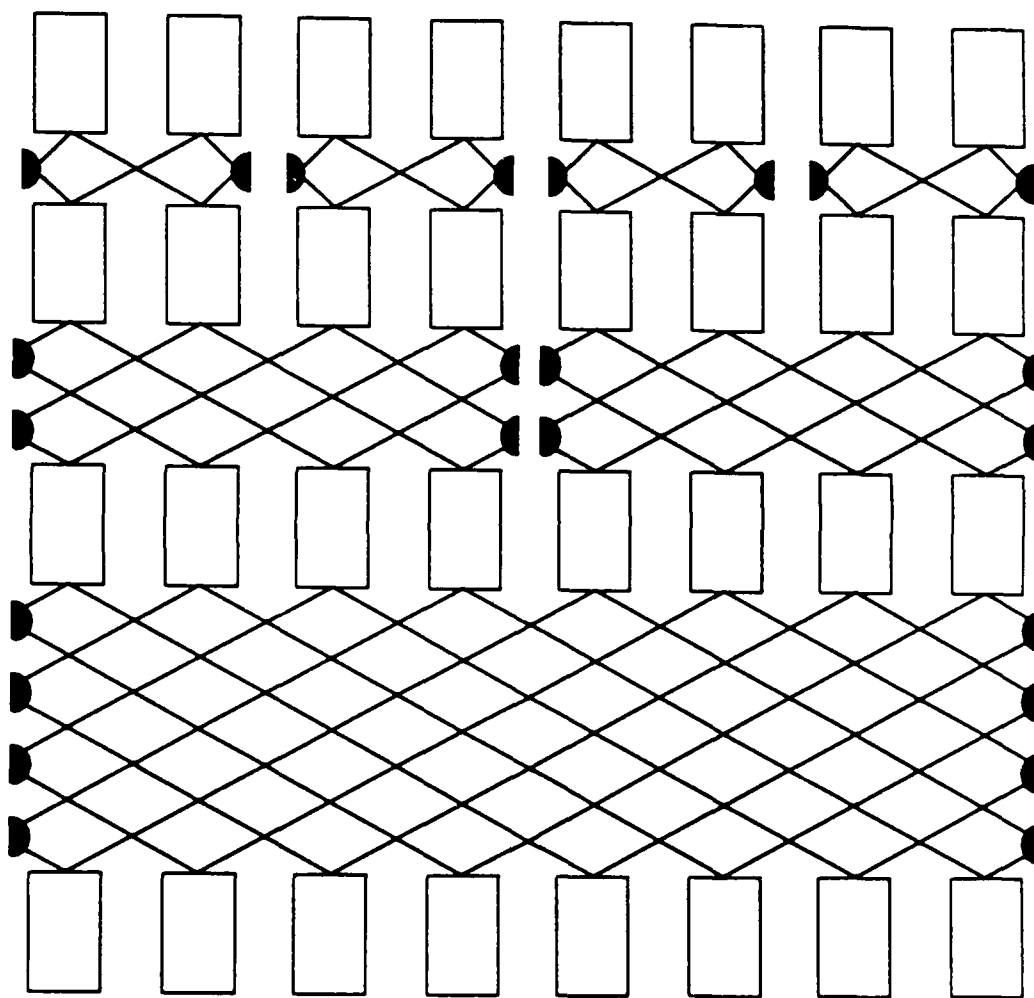


Figure 7-6: The butterfly interconnection pattern proposed by David Wise. The filled semicircles are contacts.

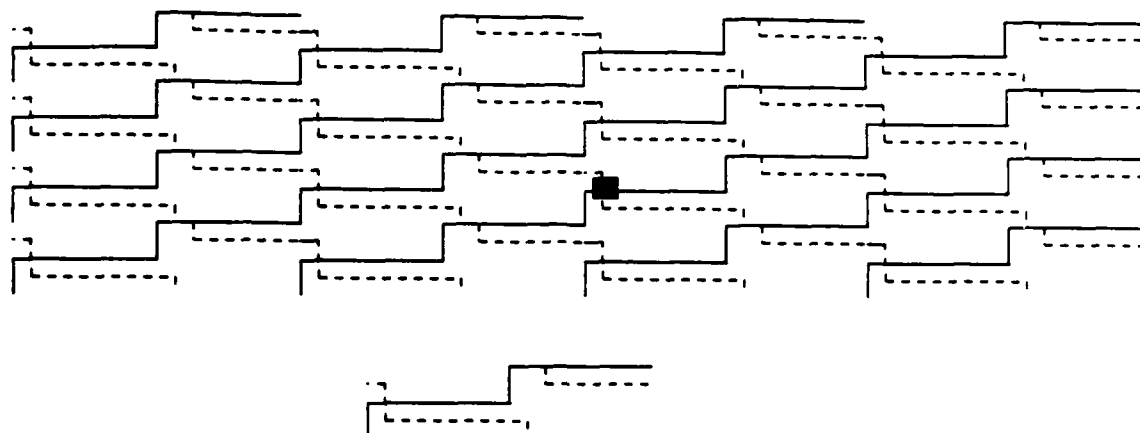


Figure 7-7: Base pattern used to implement the Wise interconnection pattern. The filled box represents one possible place for a contact. A primitive box is shown below the pattern. The figure is made of many copies of these boxes. From this pattern we can make interconnection patterns of any size by properly placing contacts and cleaning up the boundaries. The black box is a potential contact point between the dashed layer and the solid.

layer.

To see what the whole interconnect would look like, take two copies of figure 7-6. Turn one upside down (symmetry helps here) and place it on top of the other overlapping two columns. We actually did that and used the "hardcopy" to hand trace permutations during the testing of the chip.

When we were first planning this chip, we thought the switch interconnect, including the translation of the Wise layout to a rectilinear geometry, would be painful, time consuming, and error prone. We were pleasantly surprised. Although we spent a few hours planning, the methodology<sup>1</sup> we used allowed us to do the entire interconnect layout in a couple of hours with no functional errors that were not caught by design-rule check. Were we to have build a 32 bit network, it would have taken about ten to fifteen minutes longer.

First we decided to approximate the diagonal lines with many-jogged lines. It is not clear that the jogs add appreciably to resistance. Figure 7-7 illustrates the template we used to generate the network. The poly and metal lines have the same horizontal and vertical increments. The poly lines' start points are set back so that there are nice crossovers. If both types of lines started out even, then there would be long runs of direct overlap of poly and metal. This would cause fabrication difficulties because of the hill and vale problem. The poly lines may elevate the surface upon which the metal lines are placed. If a metal line runs directly over the hill of a poly line, the metal may roll off the sides.

The pattern of figure 7-7 is easy to cut into the final interconnect. The metal and poly have runs that are close together, so we just make a contact over the two as shown by the solid boxes in figure 7-7. We then remove the unnecessary lines. The runs upon which the contacts are placed are large enough for two contacts side by side, so two patterns will fit together nicely

<sup>1</sup> suggested by Charles Leiserson

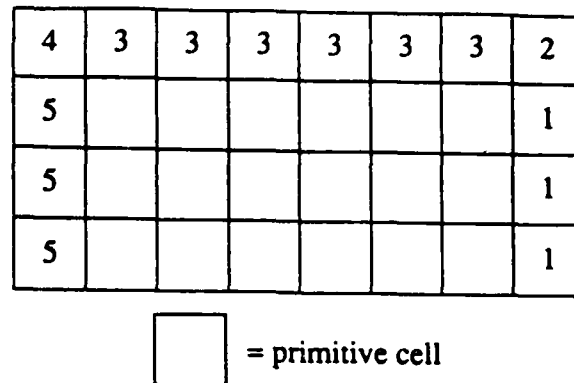


Figure 7-8: The places where we use special interconnect cells. This represents the tile pattern between columns 0 and 1. The numbers within the boxes are used as identifiers. For example, the upper right corner is a different cell from the upper left corner.

the way figure 7-6 suggests.

The actual layout was quite easy. We made a primitive box (see figure 7-7). We decided the dimensions of this box based upon the vertical spacing between outputs in a switch row and upon the minimum contact size and spacings. The primitive boxes tile to make the interconnect pattern of figure 7-6.

We started by laying out a four input by four input (2 switch by 2 switch) interconnect pattern, then an eight input by eight input pattern, and so on making special cells as needed. The special cells put in the contacts and break lines, form the connection to the inputs of the next switch row, etc. In all we needed five special cells as illustrated in figure 7-8.

We wanted to be sure that we could still tile cells to make interconnection blocks and tile blocks to make an entire connection between columns. Therefore, we added small, nonfunctional boxes of diffusion to the special cells such that the bounding boxes of the special cells aligned as needed. We knew we could ignore any design rule violations that had to do with diffusion because the interconnect pattern uses only poly and metal. We removed the diffusion tabs when the switch interconnect was completely finished.

### The Poly Register

All of the combinational logic used to generate subcontrol signals looks very similar. All subcontrol signals are generated in pairs—the true and complement—and all pass through superbuffers. We created two cells, each of which buffers a signal and its complement. We also built an inverter for the signal that fits right in front of the buffer cells. In the poly register control, both pairs of signals generated have the input to the inverter routed high and the result of the inversion routed low. However, the interface between the buffers and the inverter allows routing the other way. The other control units sometimes do that. The superbuffers tile well and the nand gate rails match those of the buffers and the inverter.

We used the same methodology to build all control logic. If there were  $k$  (sig,  $\overline{\text{sig}}$ ) pairs to be generated, then we started with  $k$  buffer cells tiled as with the poly register control unit. Then we added the inverters and attached them to buffers depending upon the order in which we wanted the signals to emerge at the left side. The far right end (signal generation) depended upon the exact logic necessary. There are 2-, 3-, and 4-input nands, all with rails to match

the buffers and inverters. Most of the interconnect is either straight across a rail, or travels along the rail.

The individual poly register cells are pitch-aligned with the main register cells which are in turn aligned with the switch columns. The aligning was somewhat difficult because the switches are not evenly spaced. The space needed for the interconnection pattern between columns grows exponentially outward from the center of the network. As shown in figure 7-6, the connection between the innermost columns requires one contact, the next interconnection requires two, then four, etc. Fortunately, we did not have to hand route the interconnection of poly register cells. The broadcast control signals and vdd rail do not jog within the cells. Therefore, we made an interconnect cell for the largest distance separating cells and reused it for all connections. Overlap does not interfere with the cell. It just coincides with the lines already there.

### The Main Register

For all the cells discussed so far, the subcontrol signals were for simple load/hold registers and the function was clear. With the main register, however, a little explanation of the sub-control signals is in order. The signals c-load, c-shift, and c-feedback control what type of value is loaded into the main register. Exactly one of the three signals is high on each  $\phi_1$ . They are all low when  $\phi_1$  is low since the main register only loads new values on  $\phi_1$ . If c-load is high, the new value is from off-chip. If c-shift is high, the new value of each cell comes from its right neighbor (or 0 at the far right). If c-feedback is high, the new value is a function of the old value. The three signals c-xornorth, c-xorsouth, and c-xorgnd control the value of the feedback loop (i. e. the value loaded if c-feedback is high). Again, only one can be high at once. If c-xornorth is high, the value in the feedback path is the old value xor'd with the corresponding bit of the poly register. If c-xorsouth is high, the value of the feedback path is the old value xor'd with the corresponding value from the switch network. If c-xorgnd is high, the feedback is the old value (a straight hold).

The preceding section discussed some of the methodology used in the layout of the main register control unit.

### 7.6.3 Problems with the Layout

The major problem with this chip is that the global placement and routing is, to be generous, poor. For example, there are large bundles of poly wires which run three quarters of the way around the perimeter of the design. When we originally drew the floorplan, we had very little idea of the sizes of each of the pieces. We wanted the control unit on the left side near the most significant bit of the main register which is needed as an input to the pla. We did not think the combinational logic for the switches would be as small as it was. We also chose to send the end-around signals in poly because we thought we would be squeezed for routing space. By the time the true sizes were apparent, our hands were tied. In other words, we did not want to take the time and risk involved in changing a working layout and replanning the layers and directions of most of the signals.

There are many ways the layout can be improved. If the placement of the units stay the same, the routing can improve. First we can change the long poly end around runs to metal and make the signals to/from the pins that cross these signals do one jump to poly and back. The external-load signals have to jump the ground lines anyway. Another possibility would be

to rotate the combinational logic for the switches and route the signals around the left end. This would make connections to the control unit harder to route, however.

A better move would be to move the combinational logic for the switches underneath the design. Then we could route the two control signals around to the left, the short way, rather than making eight long runs. This would pose a new problem of what to do about inverting the clocks. Perhaps we could still invert  $\phi_2$  within the combinational logic for the switch, connecting to the  $(\phi_2, \overline{\phi_2})$  lines that run within a column. The input latch vector might be a good place to invert  $\phi_1$ , as far as layout is concerned, but that might lead to a big skew between  $\phi_1$  and  $\overline{\phi_1}$  within the pla since the pla would be attached to the  $\phi_1$  pin. The true signal would travel from the pla to the latches where it would be inverted. Then the complement would have to travel back to the pla, arriving after the signal from the pin.

One other possible fix would be to move the combinational logic for the switches below as just explained, shift the pla slightly to the right, and move the subcontrol generation units for the poly register and main register to the left end rather than the right. Then the control signals for the two registers would not have to be bused from the pla across the whole chip. Anyway, the possibilities are endless and most of them are better than what is there.

Minimum size inverters have many problems, and in fact 2:1 inverters are much better. This optimization, however, may make the chip too large for the pad frames we have available.

# Bibliography

- [1] A. Agrawal, G. E. Blelloch, R. L. Krawitz, and C. A. Phillips, "Four vector-matrix primitives," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1989, pp. 292-302.
- [2] R. Aleliunas, "Randomized parallel communication," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1982, pp. 60-72.
- [3] W. Alexi, B. Chor, O. Goldreich, and C. Schnorr, "RSA/Rabin bits are  $1/2 + 1/\text{poly}(\log n)$  secure," *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1984, pp. 449-457.
- [4] D. P. Ahlfeld, R. S. Dembo, J. M. Mulvey, and S. A. Zenios, "Nonlinear programming on generalized networks", *ACM Transactions on Mathematical Software*, Vol. 13, No. 4, 1987, pp. 350-368.
- [5] E. Bach, "How to generate random integers with known factorization," *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* 1983, pp. 184-188.
- [6] K. E. Batcher, " 'STARAN' parallel processor system hardware," *AFIPS Conference Proceedings*, Vol. 43, pp. 405-410.
- [7] H. Beker and F. Piper, *Cipher Systems*, Northwood Books, London, 1982.
- [8] J. L. Bentley, "A case study in applied algorithm design: The traveling salesman problem." unpublished manuscript, April 1983.
- [9] J. L. Bentley and J. B. Saxe, "An analysis of two heuristics for the Euclidean traveling salesman problem," *Proceedings of the Eighteenth Annual Allerton Conference on Communications, Control, and Computing*, Monticello, IL, October 1980, pp. 41-49.
- [10] D. P. Bertsekas and D. El Baz, "Distributed asynchronous relaxation methods for convex network flow problems.", *SIAM Journal on Control and Optimization*, Vol. 25, No. 1, 1987, pp. 74-85.
- [11] D. P. Bertsekas, "The auction algorithm: a distributed relaxation method for the assignment problem," In R.R. Meyer and S.A. Zenios (eds.), *Parallel Optimization on Novel Computer Architectures.*, *Annals of Operations Research*, Vol. 14, 1988.

- [12] D.P. Bertsekas, P. Hossein, and P. Tseng, "Relaxation methods for network flow problems with convex arc costs", *SIAM Journal on Control and Optimization*, Vol 25, 1987, pp. 1219-1243.
- [13] T. O. Binford, "Survey of model-based image analysis systems," *The International Journal of Robotics Research*, Vol. 1, No. 1, 1982, pp. 18-64.
- [14] G. E. Blelloch, "Parallel prefix vs. concurrent memory access," Technical Report, Thinking Machines Corporation, 1986.
- [15] G. E. Blelloch, "Applications and algorithms on the Connection Machine," *TR87-1*, Thinking Machines Corporation, Cambridge, MA, November 1987.
- [16] G. E. Blelloch, "The scan model of parallel computation", In *Proceedings of the International Conference on Parallel Processing*, 1987.
- [17] G. E. Blelloch, *Scan Primitives and Parallel Vector Models*, PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, November 1988.
- [18] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo-random bits," *SIAM Journal of Computing*, Vol. 13, No. 4, Nov. 1984, pp. 850-864.
- [19] A. Borodin and J.E. Hopcroft, "routing, merging, and sorting on parallel models of computation," *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, 1982, pp. 338-344.
- [20] J. Boyar and H. Karloff, "Coloring planar graphs in parallel," *Journal of Algorithms*, Vol. 8, 1987, pp. 470-479.
- [21] D. Burton, *Introduction to Modern Abstract Algebra*, Addison-Wesley, 1967.
- [22] P. R. Capello, "Gaussian elimination on a hypercube automation," *Jf. Parallel and Dist. Comput.*, Vol. 4, pp. 288-308, July 1987.
- [23] V. Cherkassky and R. Smith, "Efficient mapping and implementation of matrix algorithms on a hypercube," *The Journal of Supercomputing*, Vol. 2, No. 1, pp. 7-27, September 1988.
- [24] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *Annals of Mathematic Statistics*, Vol. 23, 1952, pp. 493-507.
- [25] F. Chin, J. Lam, and I. Chen, "Efficient parallel algorithms for some graph problems," *Communications of the ACM*, Vol. 25, No. 9, September 1982, pp. 659-665.
- [26] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree, and graph problems," *Proceedings of the 27th Annual IEEE Symposium on Foundation of Computer Science*, Oct. 1986, pp. 478-491.
- [27] S. A. Cook, "A taxonomy of problems with fast parallel solutions," *Information and Control*, Vol. 64, pp. 2-22.

- [28] S. A. Cook, C. Dwork, and R. Reischuk, "Upper and lower time bounds for parallel random access machines without simultaneous writes," *SIAM Journal of Computing*, vol. 15, 1986, pp. 87-97.
- [29] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1989, to appear.
- [30] H. S. M. Coxeter, *Introduction to Geometry*, Wiley and Sons, NY, 1961.
- [31] H. S. M. Coxeter, *Projective Geometry*, University of Toronto Press, Toronto, 1974.
- [32] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," Technical Report 87-1, Tufts University, January 1987.
- [33] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.
- [34] R.S. Dembo, J.M. Mulvey, and S.A. Zenios, "Large scale nonlinear network models and their application", Report EES-86-18, Civil Engineering and Operations Research, Princeton University, 1986.
- [35] S. R. Deshpande and R. M. Jenevin, "Scaleability of a binary tree on a hypercube." In *1986 International Conference on Parallel Processing*, pp. 661-668, IEEE Computer Society, 1986.
- [36] P. Dymond and W. L. Ruzzo, "Parallel RAMs with owned global memory and deterministic context-free language recognition," *Proceedings of the Thirteenth International Colloquium on Automata, Languages, and Programming*, 1986, pp. 95-104.
- [37] D. M. Eckstein, "Simultaneous memory access," Technical Report TR-79-6, Computer Science Department, Iowa State University, Ames, Iowa, 1979.
- [38] H. Graham Flegg, *From Geometry to Topology*, The English Universities Press Ltd., distributed in the United States by Crane, Russak and Company, New York, NY, 1974.
- [39] F. E. Fich and A. Widgerson, "Towards understanding exclusive read," *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1989, pp. 76-82.
- [40] I. S. Filotti, F. L. Miller, and J. Reif, "On determining the genus of a graph in  $O(v^{O(g)})$  steps," *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, 1979, pp. 27-37.
- [41] S. Fortune and J. Wyllie, "Parallelism in random access machines," *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing*, 1978, pp. 114-118.
- [42] G. C. Fox, S. W. Otto, and A. J. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication," *Parallel Computing*, Vol. 4, No. 1, pp. 7-31, 1987.
- [43] G. C. Fox and W. Furmanski, "Optimal communication algorithms on hypercube," *Technical Report CCCP-314*, California Inst. of Technology, Pasadena, CA, July 1986.

- [44] E. Gafni, J. Naor, and P. Ragde, "On separating the EREW and CROW models", to appear in *Theoretical Computer Science*.
- [45] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *Proceedings of the 27th Annual IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 492-501.
- [46] H. Gazit and J. Reif "A deterministic EREW parallel algorithm for finding the connected components in a graph," unpublished manuscript, 1988.
- [47] G. A. Geist and M. T. Heath, "Matrix factorization on a hypercube multiprocessor," In *Proceedings of the First Conference on Hypercube Multiprocessors*, pp. 161-180, August 1985.
- [48] A. V. Goldberg, S. A. Plotkin, and G. Shannon, "Parallel symmetry breaking in sparse graphs," *SIAM Journal of Discrete Math*, to appear.
- [49] A. V. Goldberg, *Efficient Algorithms for Parallel and Sequential Computers*, PhD thesis, Massachusetts Institute of Technology, Jan. 1987.
- [50] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1984, pp. 464-479.
- [51] L. M. Goldschlager, "A unified approach to models of synchronous parallel machines," *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing*, 1978, pp. 89-94.
- [52] S. Goldwasser, S. Micali, and P. Tong, "Why and how to establish a private code on a public network," *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982, pp. 134-144.
- [53] M. J. Greenberg and J. R. Harper, *Algebraic Topology: A First Course*, Addison-Wesley, New York, NY, 1981.
- [54] T. Hagerup, "Optimal parallel algorithms for planar graphs," *Proceedings of AWOC*, 1988.
- [55] T. Hagerup, "Optimal parallel algorithms for planar graphs," *Information and Computation*, to appear.
- [56] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1962.
- [57] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
- [58] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM*, Vol. 29, No. 12, December 1986, pp. 1170-1183.
- [59] D. S. Hirschberg, "Fast parallel sorting algorithms," technical report, Department of Electrical Engineering, Rice University, Houston, TX, Jan. 1977.
- [60] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Communications of the ACM*, Vol. 22, No. 8, August 1979, pp. 461-464.

- [61] B. K. P. Horn, *Robot Vision*, MIT Press, Cambridge, MA, 1986.
- [62] S. W. Hornick and F. P. Preparata, "Deterministic P-RAM simulation with constant redundancy," *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 103-109.
- [63] K. E. Iverson, *A Programming Language*, Wiley, New York, 1962.
- [64] D. S. Johnson and C. H. Papadimitriou, "Performance guarantees for heuristics," in *The Traveling Salesman Problem* (Lawler, Lenstra, Kan, Shmoys, Editors), 1985, pp. 145-180.
- [65] S. L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel Distributed Comput.*, Vol. 4, No. 2, pp. 133-172, April 1987.
- [66] S. L. Johnsson and C. Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes," *Technical Report YALEU/DCS/RR-500*, Dept. of Computer Science, Yale Univ., New Haven, CT, November 1986. Revised November 1987, *YALEU/DCS/RR-610*. To appear in *IEEE Trans. Computers*.
- [67] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, Vol. 4, No. 4, 1984, pp. 373-395.
- [68] R. M Karp, "Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane," *Mathematics of Operations Research*, Vol. 2, No. 3, August 1977, pp. 209-224.
- [69] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," *Technical Report UCB/CSD 88/408*, Computer Science Divisions, University of California, Berkeley, CA, March 1988.
- [70] J. L. Kennington and R. V. Helgason, *Algorithms for Network Programming*, John Wiley, N. York, 1980.
- [71] L. G. Khachiyan, "A polynomial algorithm in linear programming," translated in *Soviet Mathematics Dohlady*, Vol. 20, No. 1, 1979, pp. 191-194.
- [72] Kishi *et. al.* (editors), *Encyclopedic Dictionary of Mathematics*, Mathematical Society of Japan, MIT Press, 1977.
- [73] D. Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1969.
- [74] V. Koubek and J. Kršňáková, "Parallel algorithms for connected components in a graph," *Proceedings of the 5th International Conference on Fundamentals of Computation Theory*, Springer Lecture Notes in Computer Science, Vol. 199, pp. 208-217.
- [75] F. T. Leighton, *Complexity Issues in VLSI*, MIT Press, Cambridge, MA, 1983.
- [76] F. T. Leighton, B. M. Maggs, and S. Rao, "Universal packet routing algorithms," *Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science*, 1988, pp. 256-269.

- [77] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, Vol. C-34, No. 10, October 1985, pp. 892-901.
- [78] C. E. Leiserson and B. M. Maggs, "Communication-efficient parallel algorithms for distributed random-access machines," *Algorithmica* Vol. 3, No. 1, 1988 pp. 53-78.
- [79] C. E. Leiserson, personal communication.
- [80] L. Levin, "One-way functions and pseudorandom generators," *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, 1985, pp. 363-365.
- [81] J. Lipson, *Elements of Algebra and Algebraic Computing*, Addison-Wesley, 1981.
- [82] W. Lim, "Fast algorithms for labeling connected components in 2-D arrays," unpublished manuscript, MIT, July 1986.
- [83] W. Lim, A. Agrawal, and L. Nekludova, "A parallel  $O(\lg n)$  algorithm for finding connected components in planar images," *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp. 783-786.
- [84] S. Lin and B. W. Kernighan, "An effective heuristic for the traveling-salesman problem," *Operations Research*, Vol. 21, pp. 498-516.
- [85] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, 1968.
- [86] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.
- [87] R. R. Meyer and S. A. Zenios, editors, *Parallel Optimization on Novel Computer Architectures*, Volume 14 of *Annals of Operations Research*, A.C. Baltzer Scientific Publishing Co., Switzerland, 1988.
- [88] G. Miller and J. Reif, "Parallel tree contraction and its application," *Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science*, 1985, pp. 478-489.
- [89] C. Moler, "Matrix computation on a distributed memory multiprocessor," In *Proceedings of the First Conference on Hypercube Multiprocessors*, pp. 161-180, August 1985.
- [90] J. R. Munkres, *Elements of Algebraic Topology*, The Benjamin/Cummings Publishing Co., Menlo Park, CA, 1984.
- [91] D. Nath and S. N. Maheshwari, "Parallel algorithms for the connected components and minimal spanning tree problems," *Information Processing Letters*, Vol. 14, pp. 7-11.
- [92] N. Nisan, "CREW PRAMs and decision trees," *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, 1989.
- [93] E. D. Nering, *Linear Algebra and Matrix Theory*, Wiley and Sons, NY, 1963.
- [94] V. Pan and J. Reif, "Efficient parallel solution of linear systems," Technical Report TR-02-85, Center for Research and Computing Technology, Aiken Computational Laboratory, Harvard University.

- [95] V. Pan and J. Reif, "Fast and efficient algorithms for linear programming and for the linear least squares problem," Technical Report TR-11-85, Center for Research and Computing Technology, Aiken Computational Laboratory, Harvard University.
- [96] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.
- [97] C. A. Phillips, "Parallel graph contraction," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architecture*, June 1989, pp. 148-157.
- [98] C. A. Phillips and S. A. Zenios, "Experiences with large-scale network optimization on the Connection Machine", in *Impact of Recent Computer Advances on Operations REsearch*, Elsevier Science Publishing Co., New York, NY, 1989.
- [99] J. L. Potter (editor), *The Massively Parallel Processor*, MIT Press, Cambridge, MA, 1985.
- [100] F. P. Preparata, "Parallelism in sorting," *Proceedings of the 1977 International Conference on Parallel Processing*, Aug. 1977, pp. 202-206.
- [101] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, 1986.
- [102] A. G. Ranade, "How to emulate shared memory," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, IEEE, October 1987, pp. 185-194.
- [103] J. Reif, "Optimal parallel algorithms for graph connectivity," Technical Report TR-08-84, Center for Research and Computing Technology, Aiken Computational Laboratory, Harvard University, 1984.
- [104] R. Rivest, personal communication.
- [105] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal of Computing*, Vol. 6, No. 3, September 1977.
- [106] S. L. Salas and E. Hille, *Calculus: One and Several Variables, Part II*, Wiley and Sons, NY, 1978.
- [107] C. Savage and J. JáJá, "Fast, efficient parallel algorithms for some graph problems," *SIAM Journal of Computing*, Vol. 10, No. 4, November 1981, pp. 682-691.
- [108] W. J. Savitch and M. Stimson, "Time bounded random access machines with parallel processing," *JACM*, vol. 26, 1979, pp. 103-118.
- [109] A. Shamir, "On the generation of cryptographically strong pseudo-random sequences," *Lecture Notes in Computer Science*, Vol. 62, Springer Verlag, 1981, pp. 544-550.
- [110] G. E. Shannon, "Reassigning tasks to processors less frequently: a technique for designing parallel algorithms," unpublished manuscript, Purdue University, October 1987.

- [111] Y. Shiloach and U. Vishkin, "An  $O(\lg n)$  parallel connectivity algorithm," *Journal of Algorithms*, Vol. 3, 1982, pp. 57-67.
- [112] M. Snir, "On parallel searching," *SIAM Journal on Computing*, Vol. 14, No. 3, 1985, pp. 688-708.
- [113] Q. F. Stout and B. Wager, "Passing messages in link-bound hypercubes," In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.
- [114] G. Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, MA, 1986.
- [115] G. Strang, *Linear Algebra and its Applications*, Academic Press, NY, 1976.
- [116] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proceedings of the 24th Annual IEEE Symposium on the Foundations of Computer Science*, 1984, pp. 12-20.
- [117] C. Thomassen, "The graph genus problem is NP-complete", *J. Algorithms*, to appear.
- [118] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.
- [119] E. Upfal, "Efficient schemes for parallel communication," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1982, pp. 55-59.
- [120] L. G. Valiant, "A scheme for fast parallel communication," *SIAM Journal of Computing*, Vol. 11, No. 2, May 1982, pp. 350-361.
- [121] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing*, 1981, pp. 263-277.
- [122] U. Vishkin, "An optimal parallel connectivity algorithm," RC9149, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1981.
- [123] U. Vishkin, "Implementation of simultaneous memory address access in models that forbid it," *Journal of Algorithms*, Vol. 4, 1983, pp. 45-50.
- [124] U. Vishkin, "Synchronous parallel computation—a survey," unpublished manuscript, Courant Institute, New York University, April 1983.
- [125] A. Waksman, "A permutation network," *Journal of the Association for Computing Machinery*, Vol. 15, No. 1, Jan. 1968, pp. 159-163.
- [126] D. Wise, "Compact layouts of banyan/FFT network," *VLSI Systems and Computations*, Editors Kung, Sproull, and Steele, Computer Science Press, 1981, pp. 186-195.
- [127] J. C. Wyllie, "The complexity of parallel computation, TR 79-387, Dept. of Computer Science, Cornell University, Ithaca, NY, 1979.

- [128] S. A. Zenios and R. A. Lasken, "Nonlinear network optimization on a massively parallel connection machine", In R.R. Meyer and S.A. Zenios (eds.), *Parallel Optimization on Novel Computer Architectures, Annals of Operations Research*, Vol. 14, 1988.
- [129] S.A. Zenios, "Parallel numerical optimization: current status and an annotated bibliography", *ORSA Journal on Computing*, Vol. 1, No. 1.
- [130] S.A. Zenios and J.M. Mulvey, "A distributed algorithm for convex network optimization problems", *Parallel Computing*, Vol. 6, 1988, pp. 45-56.
- [131] R. Zippel, personal communication.

OFFICIAL DISTRIBUTION LIST

Director 2 copies  
Information Processing Techniques Office  
Defense Advanced Research Projects Agency  
1400 Wilson Boulevard  
Arlington, VA 22209

Office of Naval Research 2 copies  
800 North Quincy Street  
Arlington, VA 22217  
Attn: Dr. Gary Koop, Code 433

Director, Code 2627 6 copies  
Naval Research Laboratory  
Washington, DC 20375

Defense Technical Information Center 12 copies  
Cameron Station  
Alexandria, VA 22314

National Science Foundation 2 copies  
Office of Computing Activities  
1800 G. Street, N.W.  
Washington, DC 20550  
Attn: Program Director

Dr. E.B. Royce, Code 38 1 copy  
Head, Research Department  
Naval Weapons Center  
China Lake, CA 93555